

Operações em Vectores

Sistematização

Operações em Vectors

Classe de exemplo

```
class MyVector {  
    private static final int DEFAULT_SIZE=???;  
    private type [] v;  
    int count;  
  
    public MyVector() {  
        v = new type[DEFAULT_SIZE];  
        count = 0;  
    }  
    void insertLast(type e);  
    void removeLast();  
    void insertAt(type e, int pos);  
    void removeAt(int pos);  
    boolean searchForward(type e);  
    boolean searchBackward(type e);  
    void bubbleSort();  
    boolean binarySearch(type e);  
}
```

Os elementos do vector são de um tipo genérico *type*.

Operações em Vectors

Classe de exemplo

```
class MyVector {  
    private static final int DEFAULT_SIZE=???;  
    private type [] v;  
    int count;  
  
    public MyVector() {  
        v = new type[DEFAULT_SIZE];  
        count = 0;  
    }  
    void insertLast(type e);  
    void removeLast();  
    void insertAt(type e, int pos);  
    void removeAt(int pos);  
    boolean searchForward(type e);  
    boolean searchBackward(type e);  
    void bubbleSort();  
    boolean binarySearch(type e);  
}
```

a variável `count` servirá para contar o número de elementos armazenados no vector `v`.

Operações em Vectors

Classe de exemplo para o estudo das operações

```
class MyVector {  
    private static final int DEFAULT_SIZE=???;  
    private type [] v;  
    int count;
```

```
    public MyVector() {  
        v = new type [DEFAULT_SIZE];  
        count = 0;  
    }
```

```
    void insertLast(type e);  
    void removeLast();  
    void insertAt(type e, int pos);  
    void removeAt(int pos);  
    boolean searchForward(type e);  
    boolean searchBackward(type e);  
    void bubbleSort();  
    boolean binarySearch(type e);  
}
```

No construtor limitamo-nos a inicializar o vector `v` e o contador de elementos `count`.

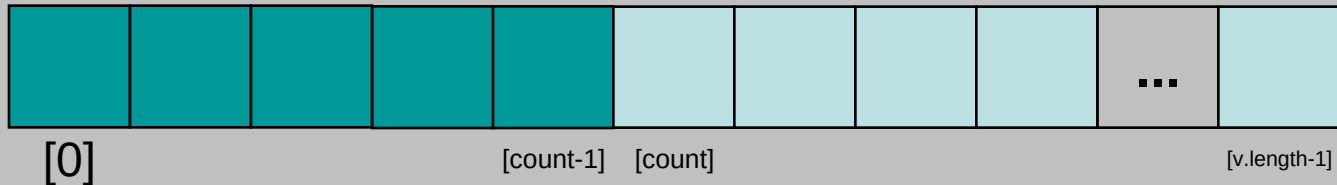
Operações em Vectors

(Inserção no final)

```
// pre: count < v.length  
void insertLast(type e) {  
    v[count] = e;  
    count++;  
}
```

Como pré-condição para a execução da operação é necessário que o vector não se encontre todo preenchido

v

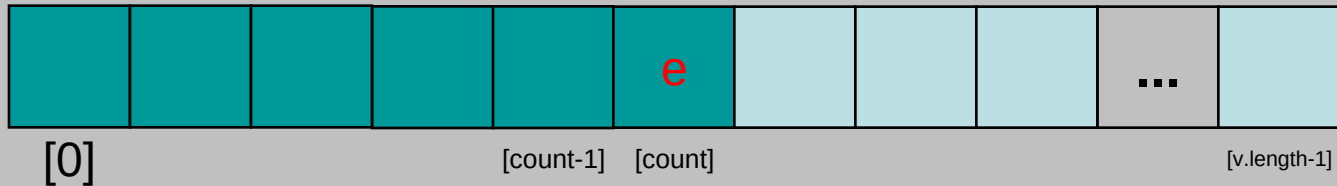


Operações em Vectors

(Inserção no final)

```
// pre: count < v.length  
void insertLast(type e) {  
    v[count] = e;  
    count++;  
}
```

v

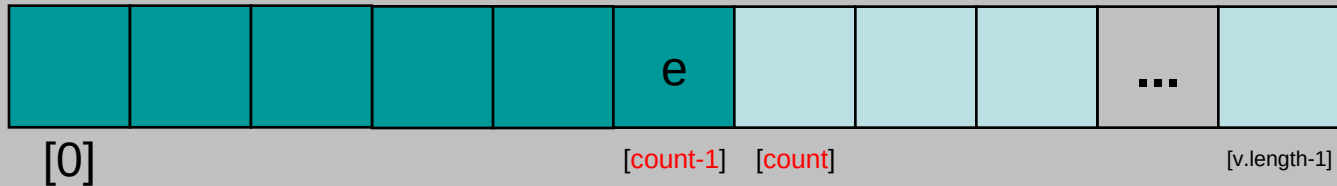


Operações em Vectors

(Inserção no final)

```
// pre: count < v.length  
void insertLast(type e) {  
    v[count] = e;  
    count++;  
}
```

v

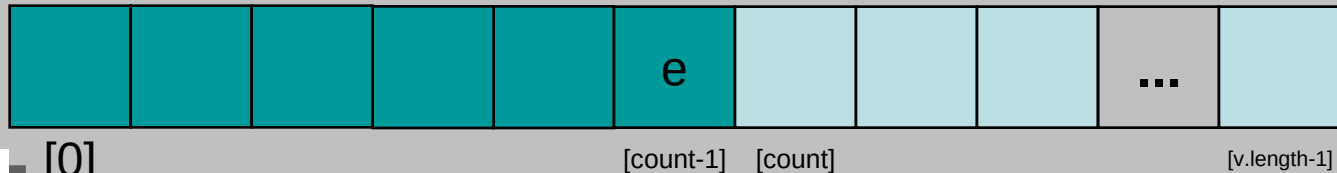


Operações em Vetores

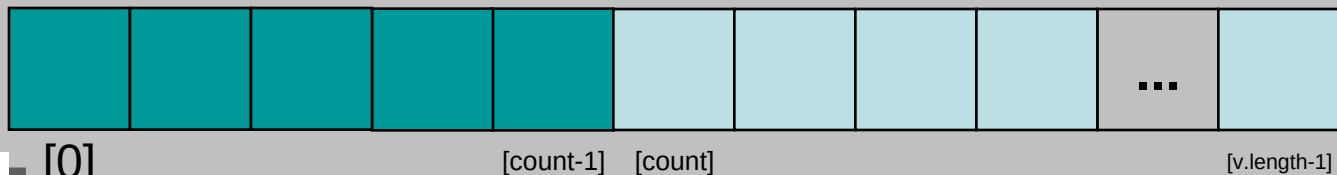
(Inserção no final)

```
// pre: count < v.length  
void insertLast(type e) {  
    v[count] = e;  
    count++;  
}
```

Depois



Antes



Operações em Vectors

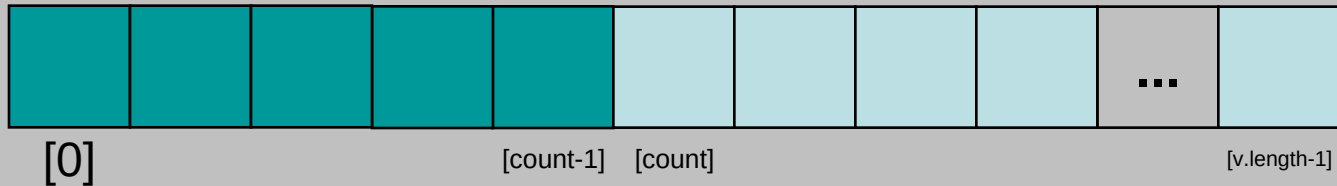
(Remoção do final)

```
// pre: count > 0
```

```
void removeLast() {  
    count--;  
}
```

O vector tem que ter pelo menos um elemento...

v



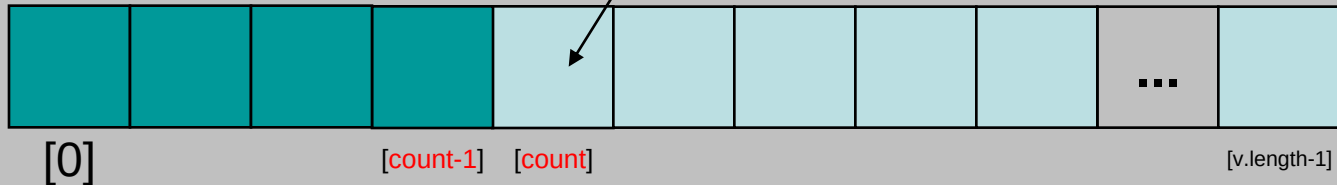
Operações em Vectors

(Remoção do final)

```
// pre: count > 0  
void removeLast() {  
    count--;  
}
```

Basta decrementar o contador de elementos! O que antes era o último elemento permanece no vector mas é “esquecido” em futuras operações...

v



Operações em Vectors

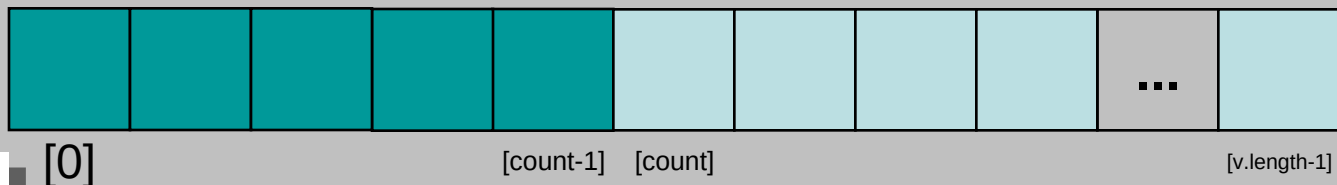
(Remoção do final)

```
// pre: count > 0  
void removeLast() {  
    count--;  
}
```

Depois



Antes



Operações em Vectors

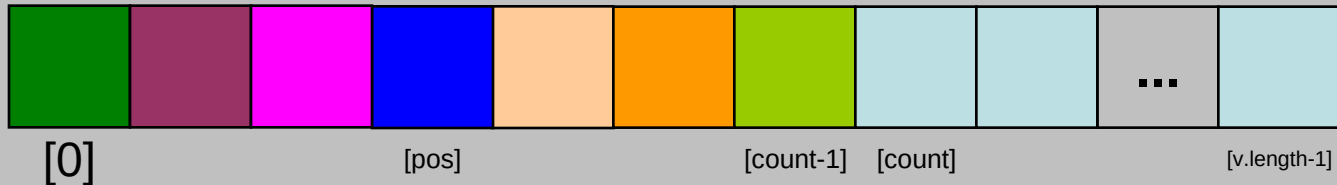
(Inserção)

```
// pre: count < v.length && pos <= count
```

```
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

O vector não pode estar todo preenchido e a posição dada tem que estar preenchida ou ser a primeira vazia

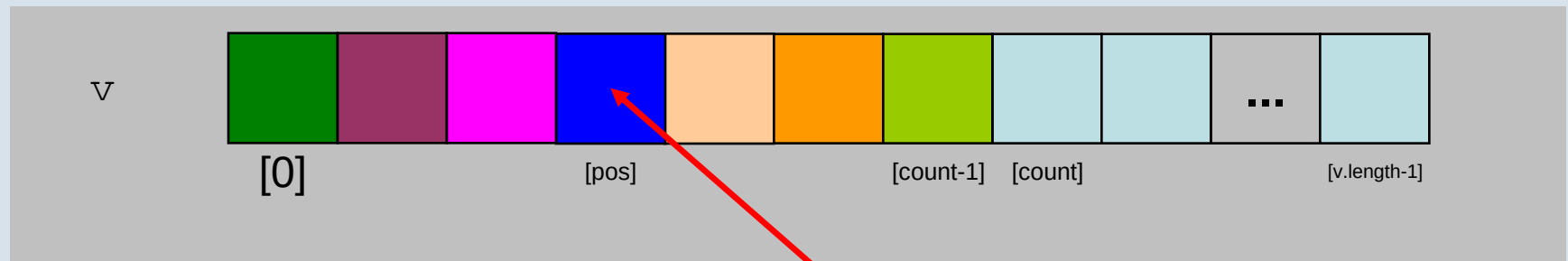
v



Operações em Vectors

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```



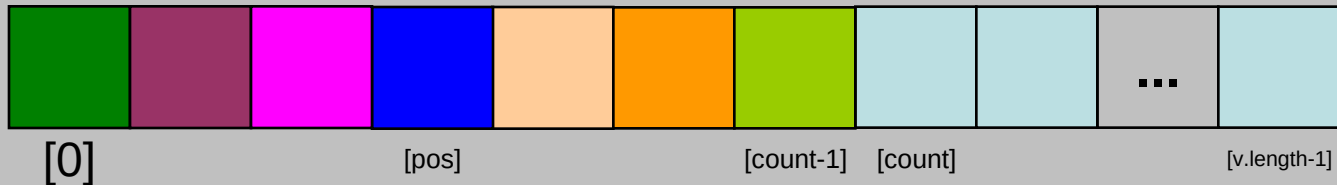
o elemento `e` deverá ser
inserido na posição de índice
`pos`.

Operações em Vectors

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

v



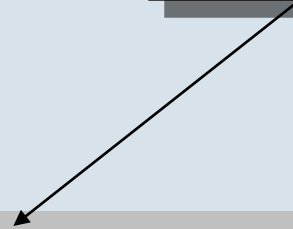
Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=count-1



Mas antes é necessário abrir espaço para o lá colocar...

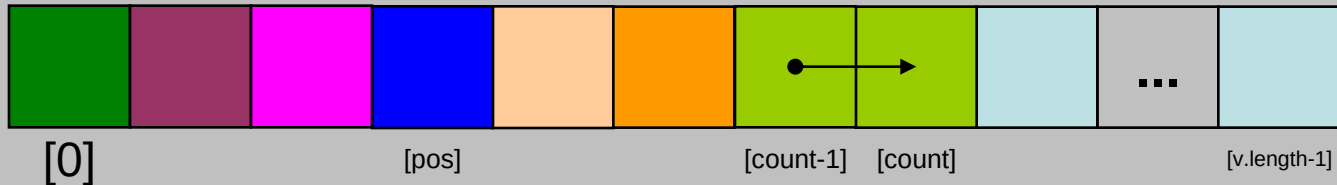
Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=count-1

v



Mas antes é necessário abrir espaço para o lá colocar...

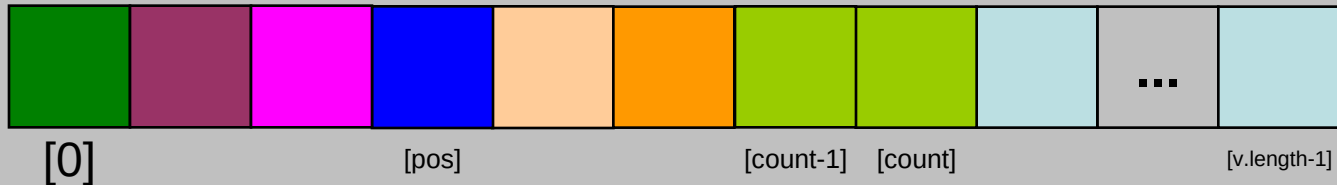
Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=count-2

v



Mas antes é necessário abrir espaço para o lá colocar...

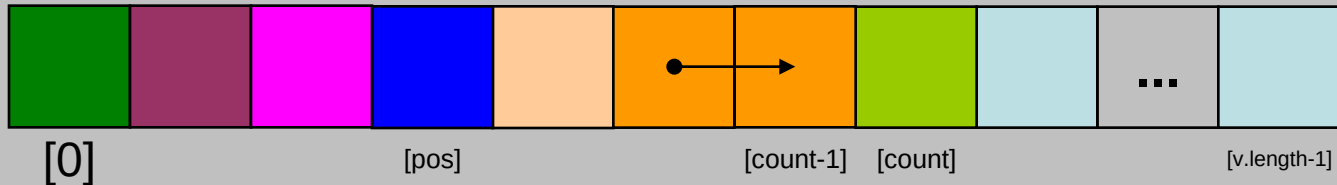
Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=count-2

v



Mas antes é necessário abrir espaço para o lá colocar...

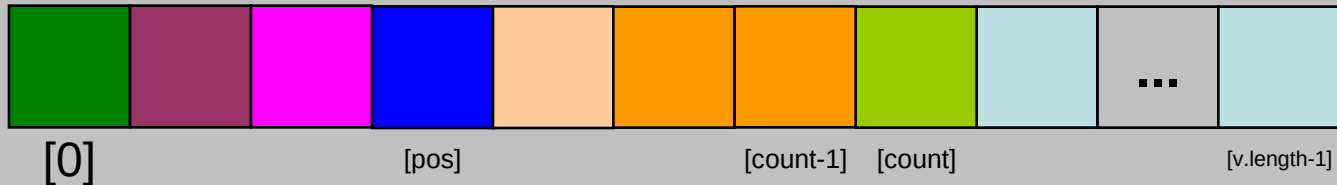
Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=pos+1

v



Mas antes é necessário abrir espaço para o lá colocar...

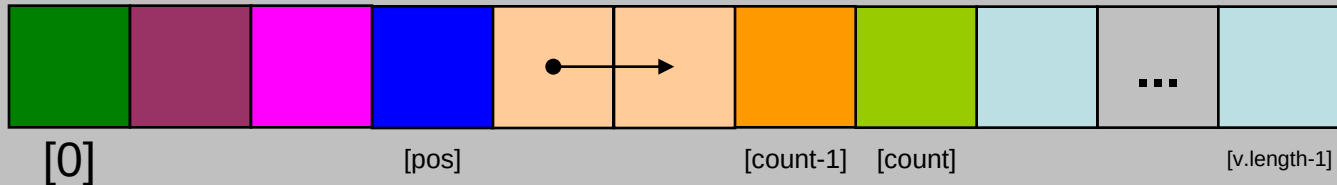
Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=pos+1

v



Mas antes é necessário abrir espaço para o lá colocar...

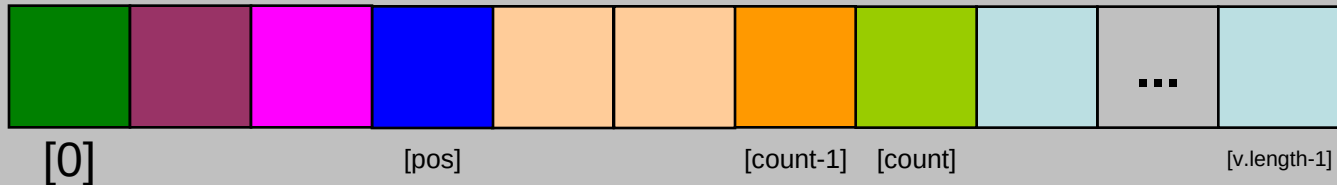
Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=pos

v



Mas antes é necessário abrir espaço para o lá colocar...

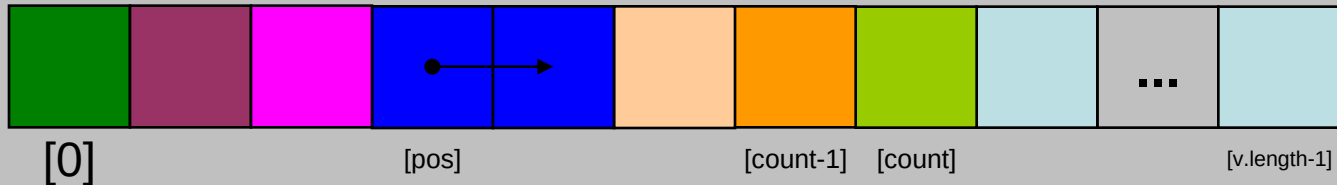
Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=pos

v



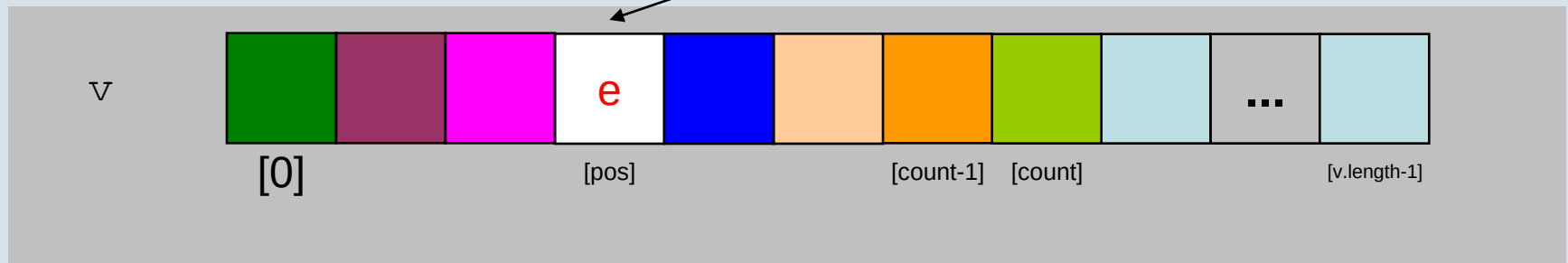
Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=pos

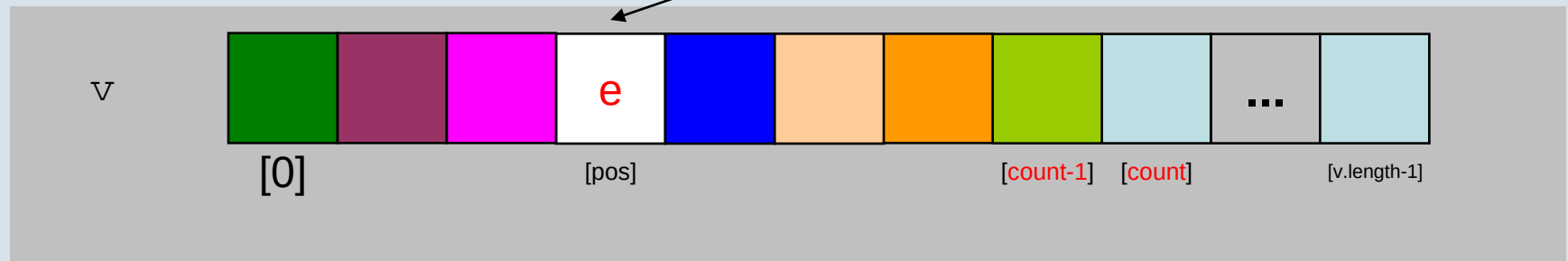


Operações em Vetores

(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

i=pos

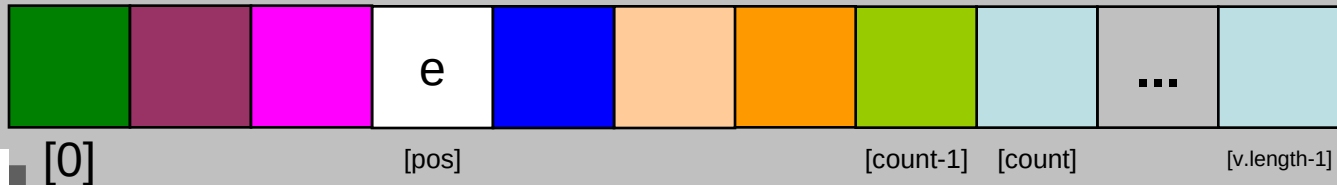


Operações em Vetores

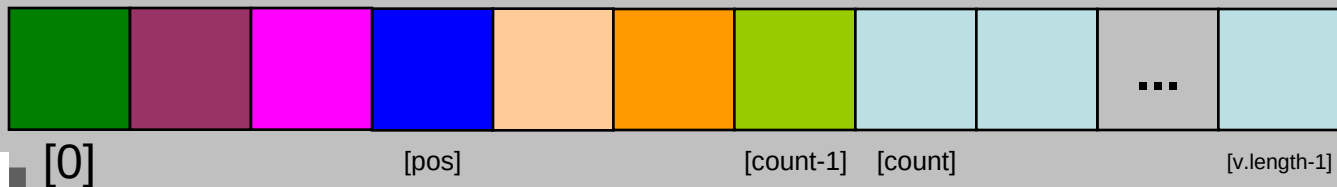
(Inserção)

```
// pre: count < v.length  
void insertAt(type e, int pos) {  
    for(int i=count-1; i >= pos; i--)  
        v[i+1] = v[i];  
    v[pos] = e;  
    count++;  
}
```

Depois



Antes



Operações em Vectors

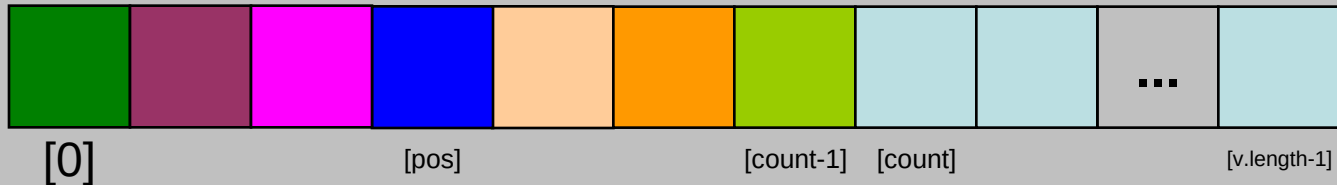
(Remoção)

```
pre: count > 0 && pos < count
```

```
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

O vector tem que ter pelo menos um elemento e a posição indicada tem que estar preenchida

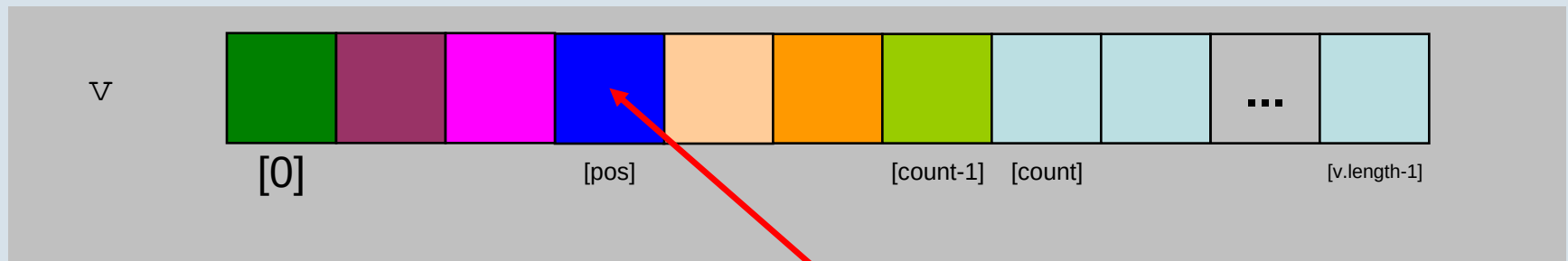
v



Operações em Vectors

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```



o elemento deverá ser
removido da posição de
índice `pos`.

Operações em Vectors

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```



Mas não podemos deixar “um buraco” no vector...

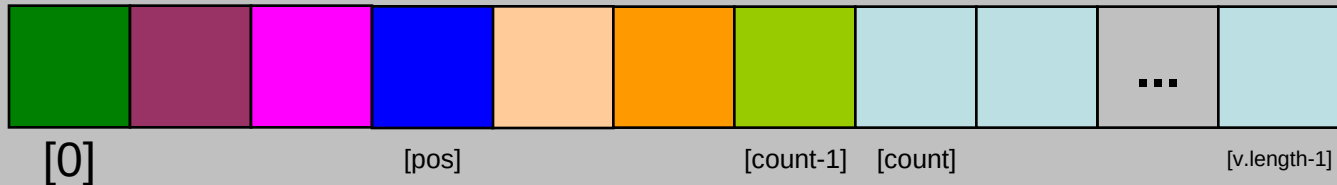
Operações em Vetores

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

i=pos

v



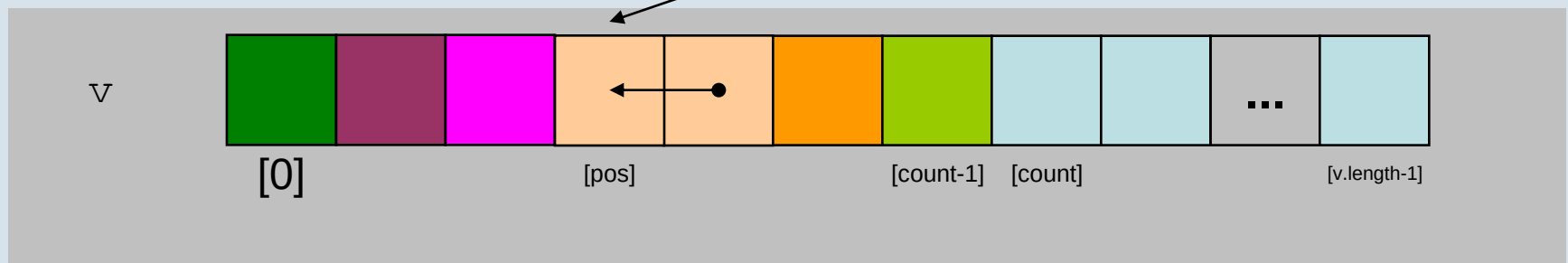
Mas não podemos deixar “um buraco” no vector...

Operações em Vetores

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

i=pos



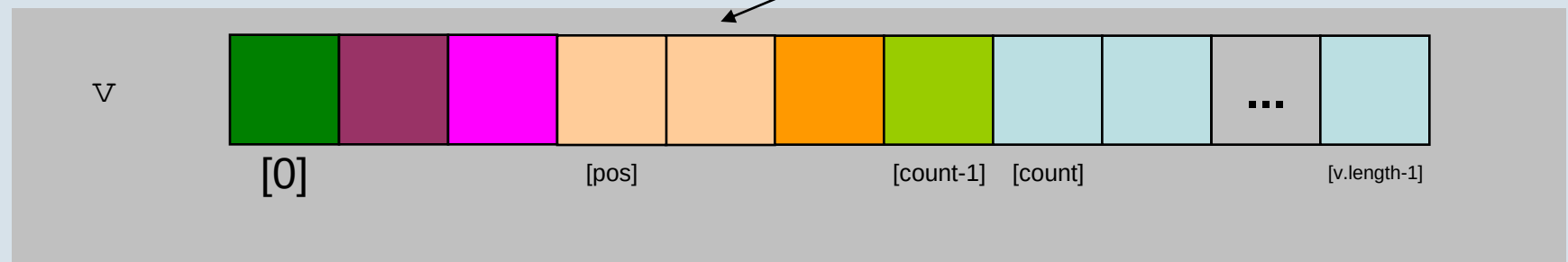
Mas não podemos deixar “um buraco” no vector...

Operações em Vectors

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

i=pos+1



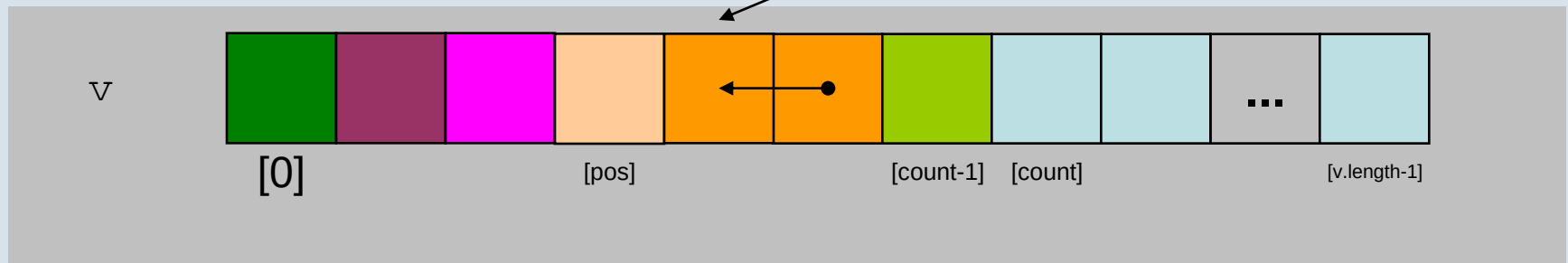
Mas não podemos deixar “um buraco” no vector...

Operações em Vetores

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

i=pos+1



Mas não podemos deixar “um buraco” no vector...

Operações em Vectors

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

i=count-2

v



Mas não podemos deixar “um buraco” no vector...

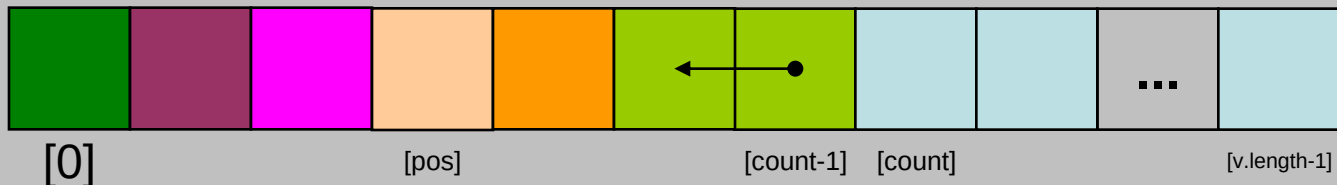
Operações em Vetores

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

i=count-2

v



Mas não podemos deixar “um buraco” no vector...

Operações em Vectors

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

i=count-1

v



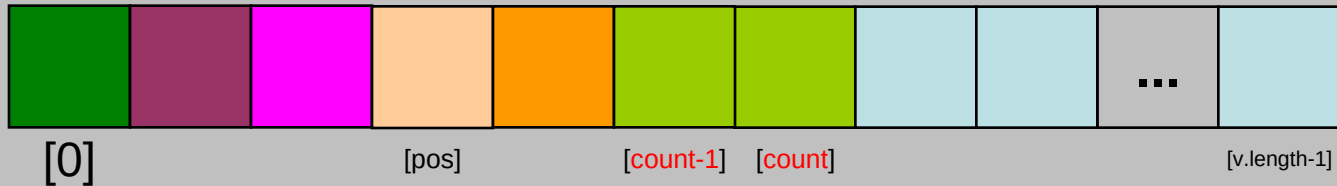
Mas não podemos deixar “um buraco” no vector...

Operações em Vectors

(Remoção)

```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

v



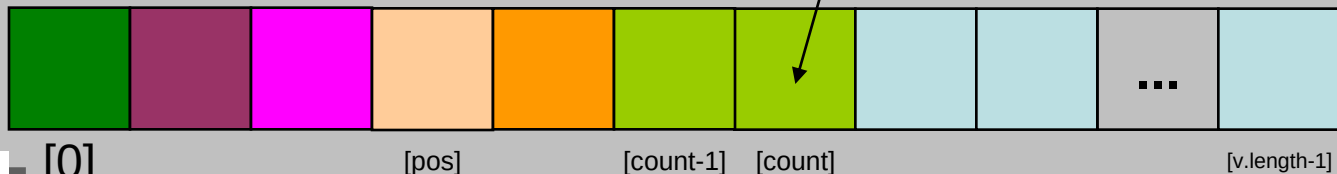
Operações em Vectors

(Remoção)

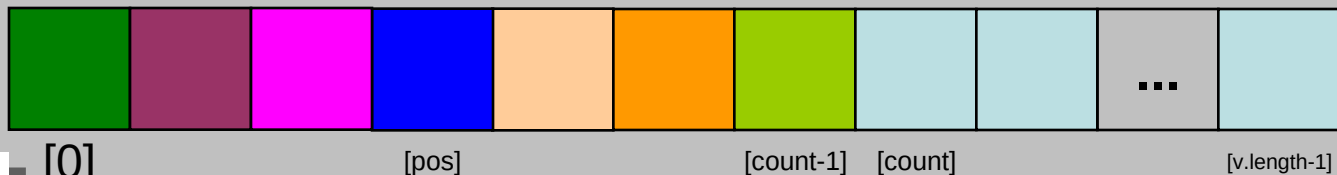
```
pre: count > 0 && pos < count  
void removeAt(int pos) {  
    for(int i=pos; i<count-1; i++)  
        v[i] = v[i+1];  
    count--;  
}
```

Apesar do último elemento estar repetido, nunca será utilizado pois não está contabilizado...

Depois



Antes

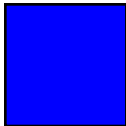


Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Existirá no vector o elemento

e =  ?

v

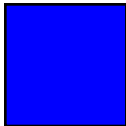


Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Existirá no vector o elemento

e =  ?

found = false

i=0

v



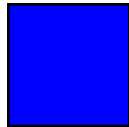
Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = false

i=0

v



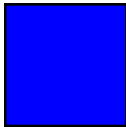
Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = false

i=0

v



Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Existirá no vector o elemento

e =  ?

found = false

i=1

v



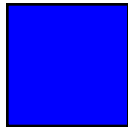
Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = false

i=1

v



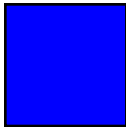
Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = false

i=1

v



Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Existirá no vector o elemento

e =  ?

found = false

i=2

v



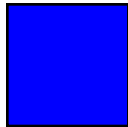
Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = false

i=2

v



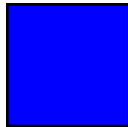
Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = false

i=2

v



Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

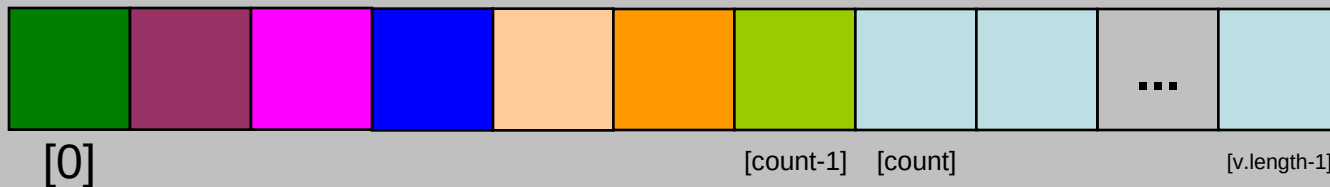
Existirá no vector o elemento

e =  ?

found = false

i=3

v



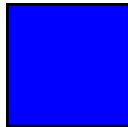
Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = false

i=3

v



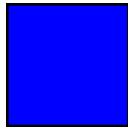
Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e) {  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = false

i=3

v



Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Existirá no vector o elemento

e =  ?

found = true

i=3

v



Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```



Existirá no vector o elemento

e =  ?

found = true

i=3

v

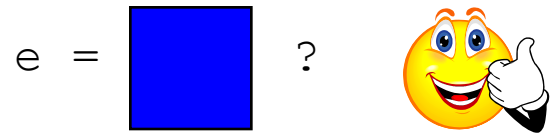


Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i] == e){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Existirá no vector o elemento



found = true

i=3

v



Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(...) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if (p(v[i])) {  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Por vezes a busca não pretende saber se um elemento específico se encontra no vector mas apenas se existe um elemento que satisfaça uma determinada condição.

$p()$ é uma função booleana e representa o critério que determina se um dado elemento é o elemento procurado.

Exemplos: procurar um elemento que seja um número primo, procurar uma conta com saldo negativo, etc...

Operações em Vectors

(Pesquisa Linear – índices decrescentes)

```
public boolean searchBackward(...) {  
    boolean found = false;  
    int i=count-1;  
    while(i>=0 && !found) {  
        if(p(v[i])) {  
            found = true;  
        }  
        else i--;  
    }  
    return found;  
}
```

Esta busca é idêntica à anterior mas efectua-se da direita para a esquerda.

Por vezes queremos saber a última ocorrência de um determinado elemento dentro dum vector...

Operações em Vectors

(Pesquisa Linear – índices crescentes)

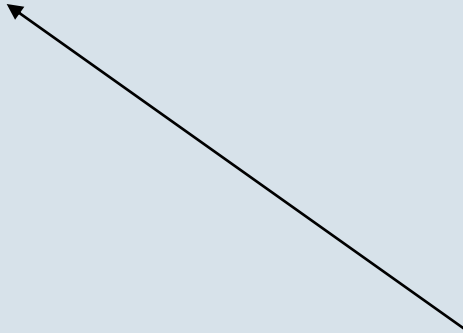
```
public int searchIndexOf(...) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(p(v[i])) {  
            found = true;  
        }  
        else i++;  
    }  
    if (found) return i;  
    else return -1;  
}
```

Se se pretender determinar a localização do elemento a procurar é fácil adaptar a função de pesquisa...

Operações em Vectors

(Pesquisa Linear – índices crescentes)

```
public boolean searchForward(...) {  
    return searchIndexOf(...) != -1;  
}
```



Mas o melhor será programar a operação mais geral (a que determina a posição do elemento dentro do vector) e programar a operação de busca simples à custa da primeira...

Operações em Vectors

(Ordenação)

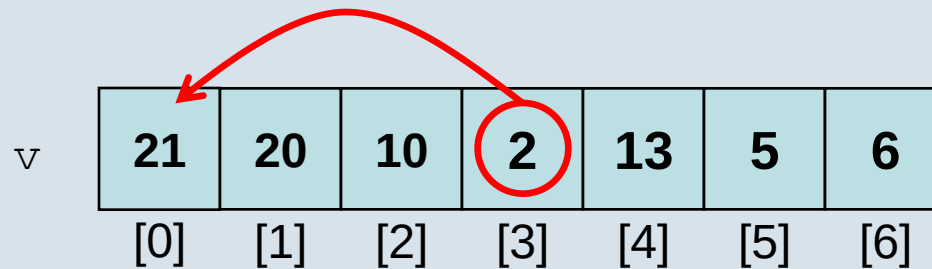
- Um dos algoritmos de ordenação mais simples é o algoritmo de Bubble Sort.
- Suponha que se pretende ordenar o seguinte vector:

v	21	20	10	2	13	5	6
	[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectors

(Ordenação)

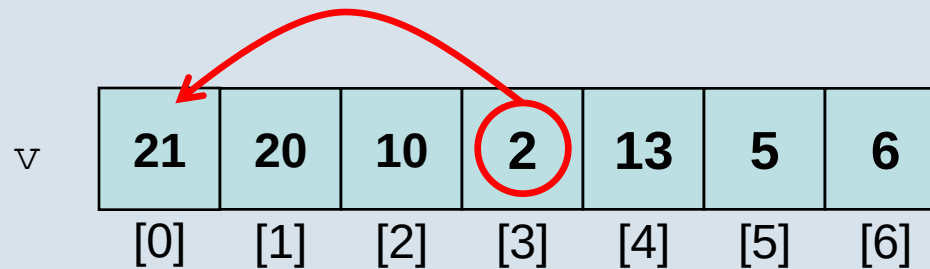
- Vamos começar por tentar empurrar o valor mais baixo para a posição mais à esquerda...
- Mas como?



Operações em Vectors

(Ordenação)

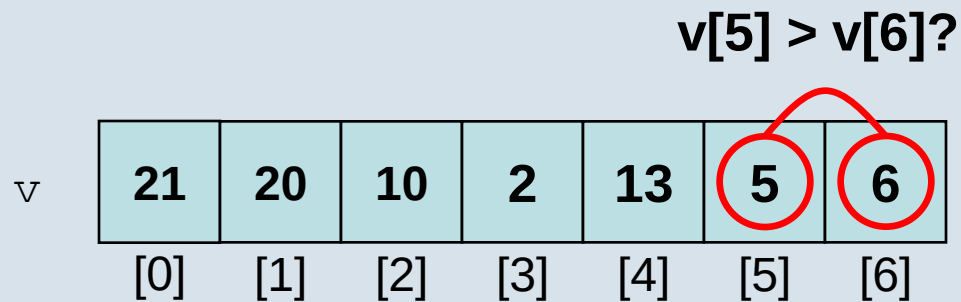
- O algoritmo Bubble Sort resolve este problema varrendo o vector da direita para a esquerda.



Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos e se estiverem fora de ordem trocam de posição



Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos e se estiverem fora de ordem trocam de posição

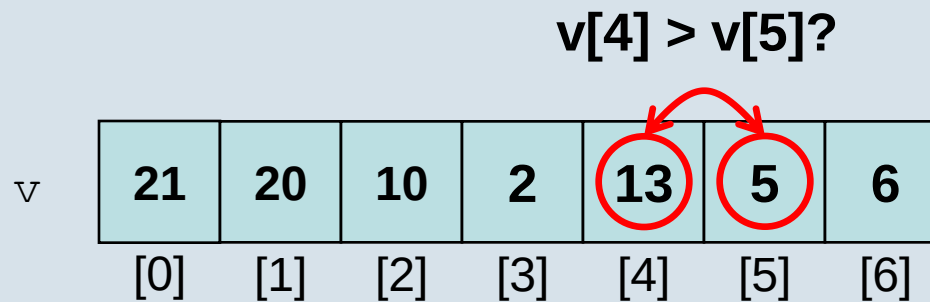
v

21	20	10	2	13	5	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda

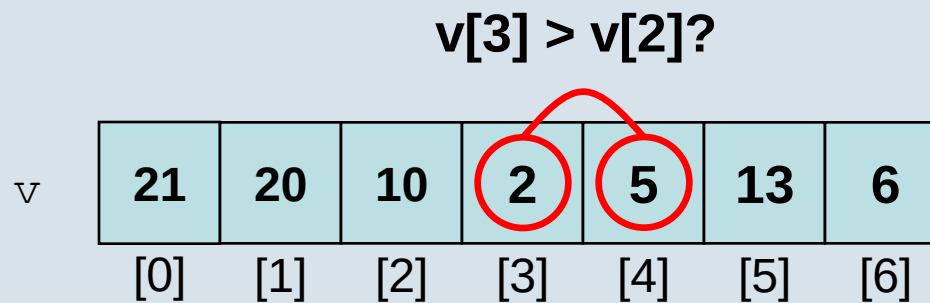
v

21	20	10	2	5	13	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda

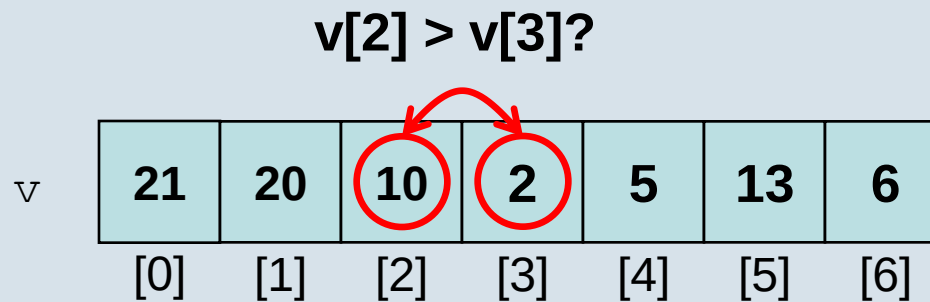
v

21	20	10	2	5	13	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda

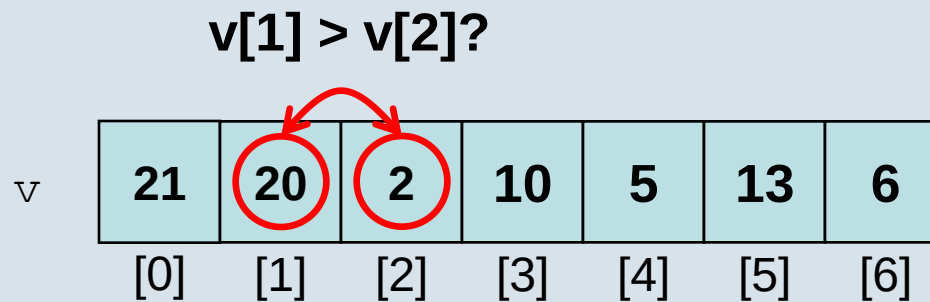
v

21	20	2	10	5	13	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda

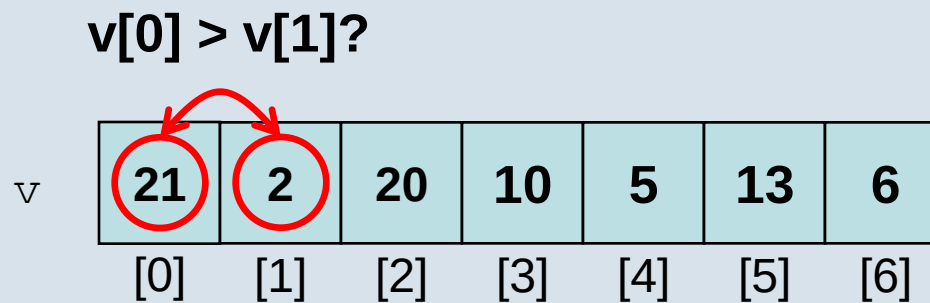
v

21	2	20	10	5	13	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectores

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda



Operações em Vectors

(Ordenação)

- O algoritmo Bubble Sort resolve o problema varrendo o vector da direita para a esquerda.
- Em cada varrimento comparam-se os elementos contíguos
- E o varrimento continua para a esquerda

v

2	21	20	10	5	13	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectors

(Ordenação)

- No final do primeiro varrimento temos o menor elemento encostado à esquerda, no seu devido lugar...

Antes

v

21	20	10	2	13	5	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Depois

v

2	21	20	10	5	13	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectores

(Ordenação)

- E estamos mais perto da solução pois já temos uma parte do vector ordenado...
- Neste caso até temos mais elementos na sua posição correcta mas foi por mero acaso 😊

Antes

v

21	20	10	2	13	5	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Depois

v

2	21	20	10	5	13	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

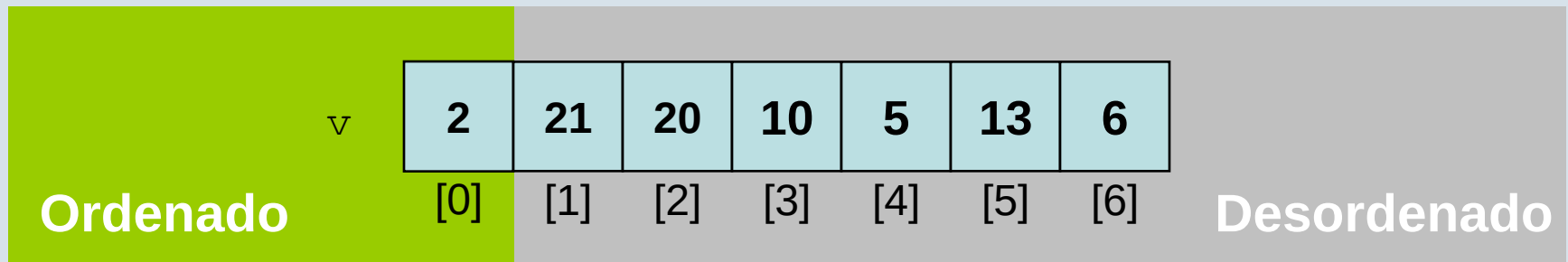
Ordenado

Desordenado

Operações em Vectores

(Ordenação)

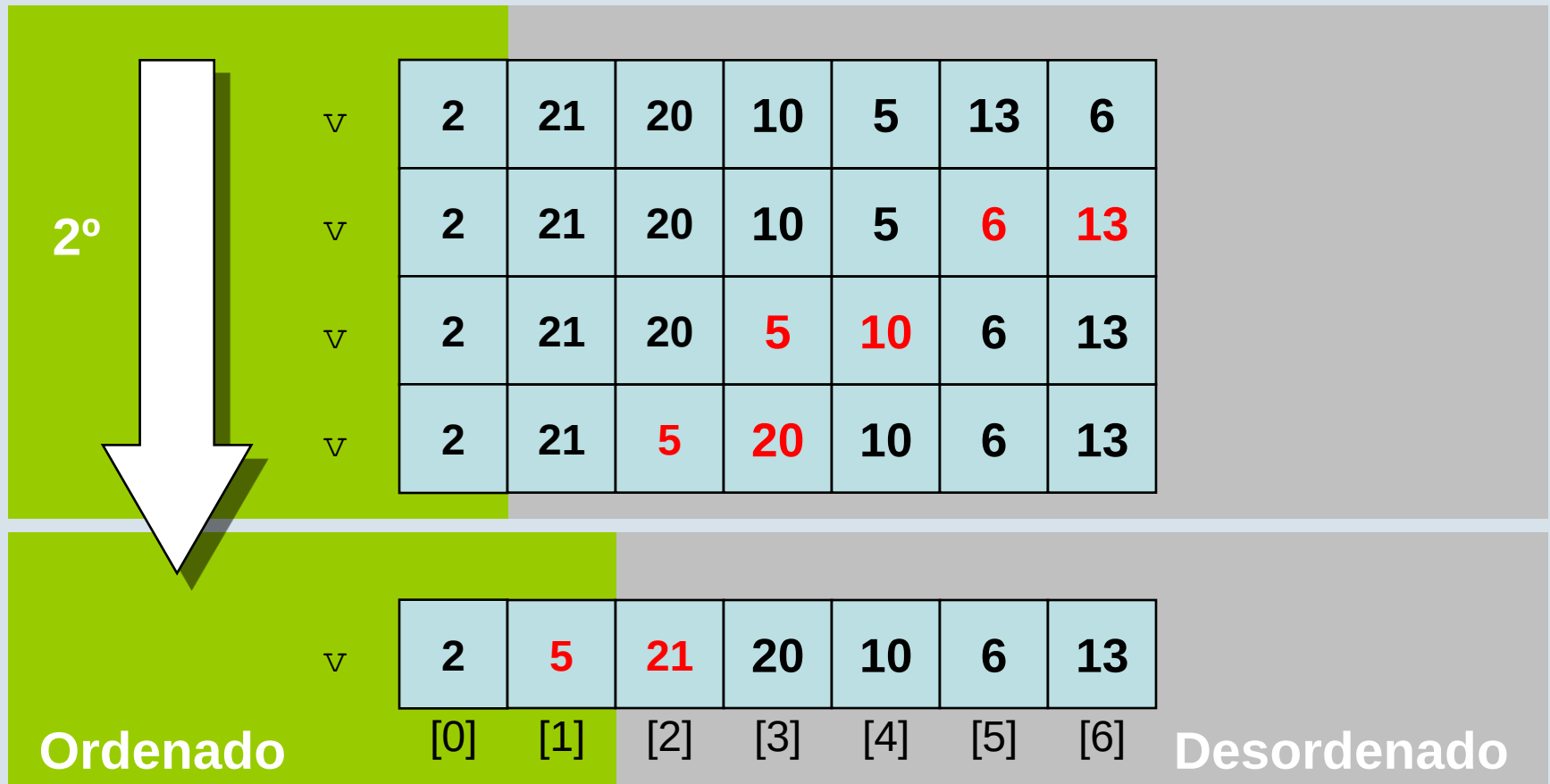
- Como fazemos para ordenar mais um elemento?
 - Basta repetir o processo mas esquecendo agora o primeiro elemento e actuando como se o vector começasse no índice 1!



Operações em Vectors

(Ordenação)

- Durante o 2º varrimento o vector passará pelos seguintes estados:



Operações em Vectors

(Ordenação)

- Durante o 3º varrimento o vector passará pelos seguintes estados:



Operações em Vectors

(Ordenação)

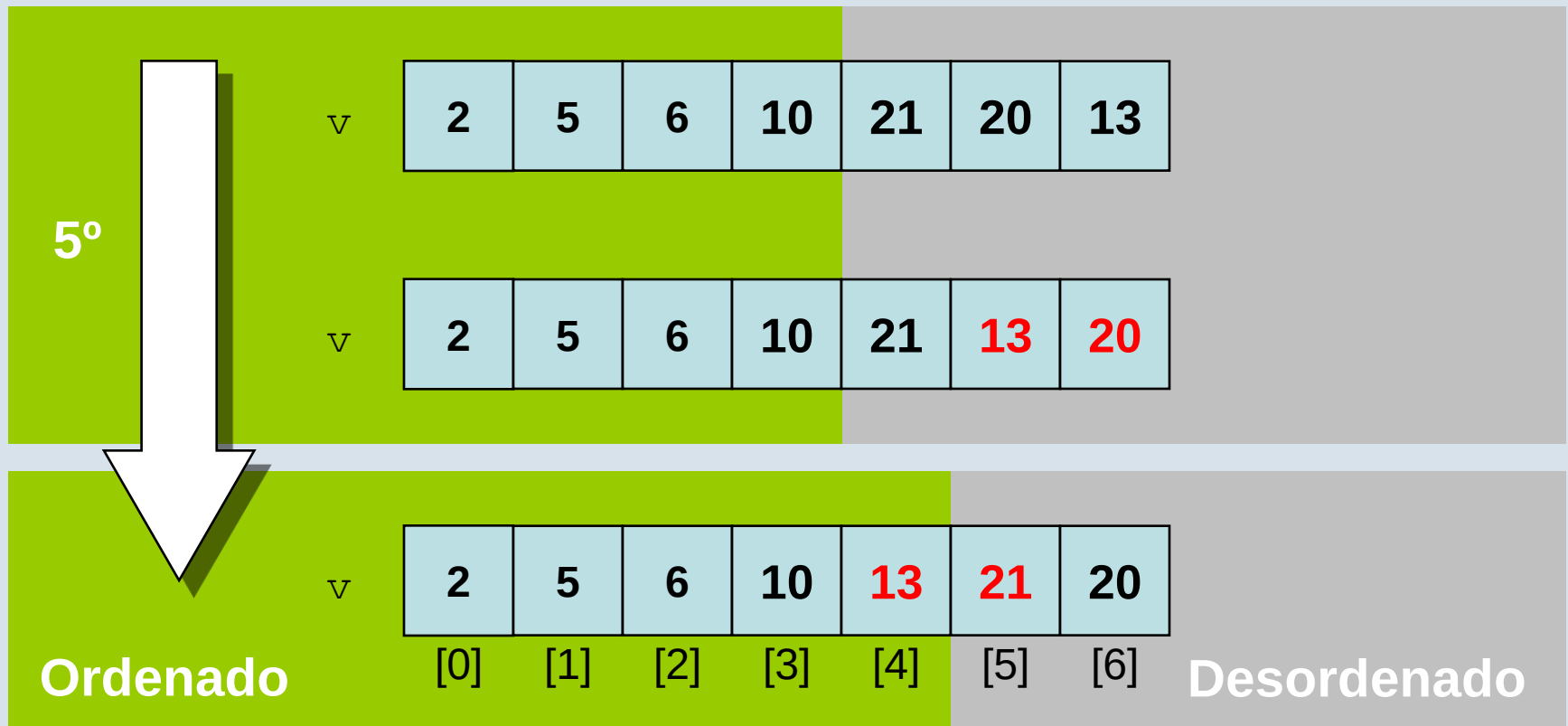
- Durante o 4º varrimento o vector passará pelos seguintes estados:



Operações em Vectors

(Ordenação)

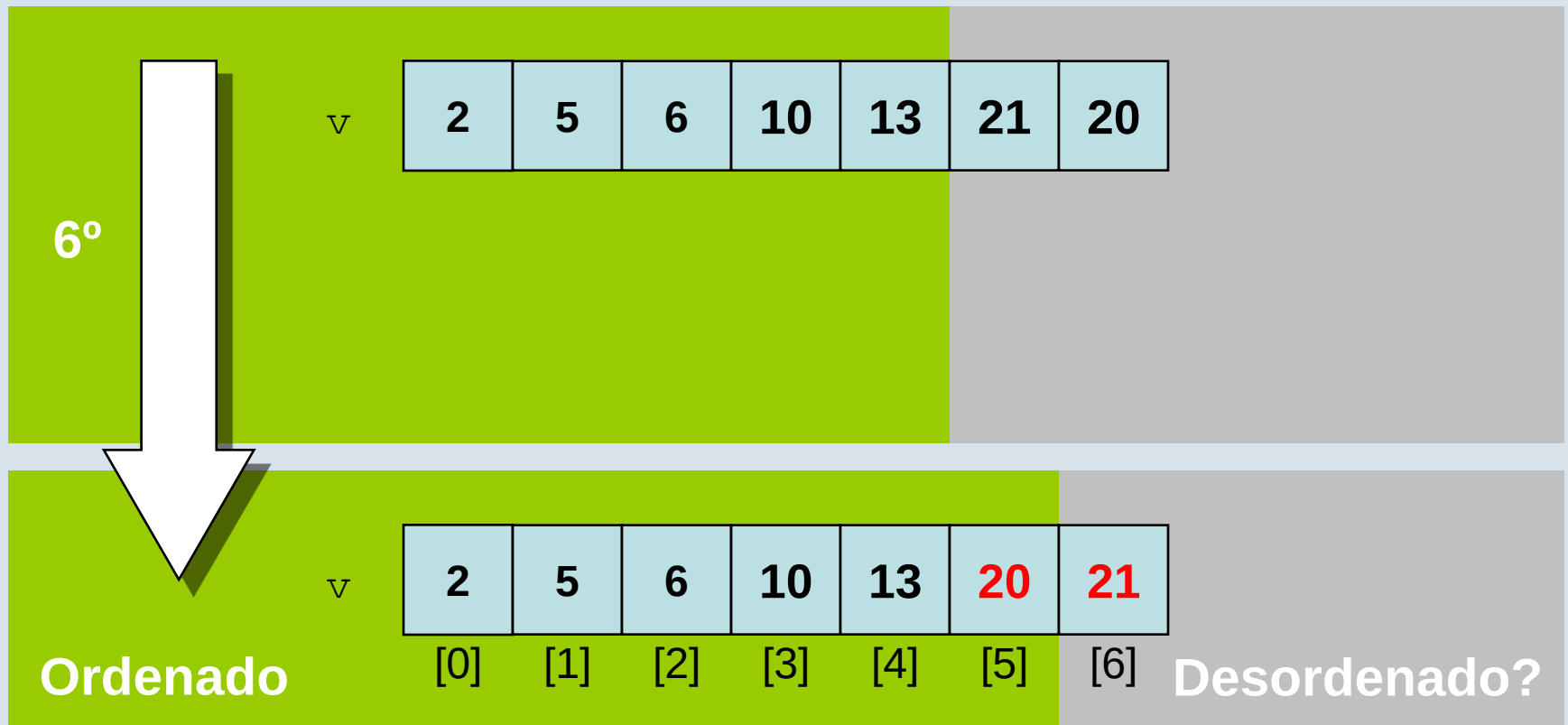
- Durante o 5º varrimento o vector passará pelos seguintes estados:



Operações em Vectors

(Ordenação)

- Durante o 6º varrimento o vector passará pelos seguintes estados:



Operações em Vectors

(Ordenação)

- Após o 6º varrimento, um vector de comprimento 7 estará ordenado!
- Para ordenar um vector de comprimento `count` serão necessários `count-1` varrimentos!

Ordenado	v	2	5	6	10	13	20	21
		[0]	[1]	[2]	[3]	[4]	[5]	[6]

Operações em Vectors

(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {
```

```
    for(int i=1; i<count; i++)
```

count-1 varrimentos

```
    ...
```

```
    // Falta programar aqui cada varrimento...
```

```
    ...
```

```
}
```

Operações em Vectors

(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {  
    for(int i=1; i<count; i++)  
        for(int j=count-1; j>=?; j--){  
            if (v[j-1] > v[j]) {  
                ...  
            }  
        }  
}
```

Cada varrimento inicia-se no final do vector

Em cada passo comparam-se dois elementos entre si.

Operações em Vectors

(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {  
    for(int i=1; i<count; i++)  
        for(int j=count-1; j>=?; j--) {  
            if(v[j-1] > v[j]) {  
  
                ...  
  
            }  
        }  
}
```

No 1º ($i=1$) varrimento, j deverá descer até ao valor 1, de modo a comparar $v[0]$ com $v[1]$.

No 2º ($i=2$) varrimento, j deverá descer até ao valor 2, de modo a comparar $v[1]$ com $v[2]$.

Operações em Vectors

(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {  
    for(int i=1; i<count; i++)  
        for(int j=count-1; j>=i; j--) {  
            if(v[j-1] > v[j]) {  
  
                ...  
  
            }  
        }  
}
```

Em cada varrimento, `j` deverá descer até ao valor de `i`!!!

Operações em Vectors

(Ordenação)

- Para ordenar um vector de comprimento `count` são necessários `count-1` varrimentos!

```
public void bubbleSort() {  
    for(int i=1; i<count; i++)  
        for(int j=count-1; j>=i; j--){  
            if(v[j-1] > v[j]) {  
                type tmp = v[j-1];  
                v[j-1] = v[j];  
                v[j] = tmp;  
            }  
        }  
    }  
}
```

Cada vez que se encontram 2 elementos fora de ordem, trocam-se de posição...

Operações em Vectors

(Ordenação)

- Eis o algoritmo de ordenação Bubble Sort:

```
public void bubbleSort() {  
    for(int i=1; i<count; i++)  
        for(int j=count-1; j>=i; j--){  
            if(v[j-1] > v[j]) {  
                type tmp = v[j-1];  
                v[j-1] = v[j];  
                v[j] = tmp;  
            }  
        }  
}
```

Pesquisa Binária

- Localiza um valor num vector ordenado, da seguinte forma:
 - Determina se o valor está na primeira ou na segunda metade do vector
 - Repete a pesquisa para uma das metades

Procurar o número 15

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {
```

O elemento a procurar (*e*), do tipo *type*, é passado como parâmetro.

```
    return ...;
```

```
}
```

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {
```

```
    boolean found=false;
```

Durante a pesquisa usaremos uma variável booleana (`found`) para indicar se já se encontrou o elemento a pesquisar.

No final, a variável deverá conter o resultado da pesquisa.

```
    return found;
```

```
}
```

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;
```

As variáveis `low` e `high` delimitam a parte do vector onde a busca incide.

No início, todo o vector deverá ser pesquisado:
`v[0]...v[count-1]`

```
    return found;
```

```
}
```

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found ..... ) {  
  
    }  
    return found;  
}
```

A pesquisa deverá prosseguir enquanto `found` valer `false`.

O mesmo será dizer:
“enquanto não encontrado”,
traduzindo a linha de código
para linguagem natural! 😊

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && ..... ) {  
  
    }  
    return found;  
}
```

Mas, no caso do elemento a pesquisar não estar contido no vector, a variável `found` nunca valerá `true`, pelo que será necessário terminar a busca com base noutra condição...

Operações em Vectores

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && low <= high) {  
  
    }  
    return found;  
}
```

No caso do elemento a pesquisar não pertencer ao vector, o intervalo de pesquisa deverá, no final, ser vazio: não há mais nenhuma metade onde procurar!

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && low <= high) {  
        int mid = (low+high)/2;  
  
    }  
    return found;  
}
```

Em cada etapa, começa-se por determinar o índice do elemento central do intervalo de busca.

Se `mid` representar o índice, `v[mid]` representa o valor central do vector.

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && low <= high) {  
        int mid = (low+high)/2;  
        if(v[mid]==e) found = true;  
        else ...  
    }  
    return found;  
}
```

Tenta-se a sorte! 😊

Poderá dar-se o caso do elemento a pesquisar estar, precisamente, nessa posição. Nesse caso assinalamos o sucesso da pesquisa.

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && low <= high) {  
        int mid = (low+high)/2;  
        if(v[mid]==e) found = true;  
        else ...  
    }  
    return found;  
}
```

Caso contrário é preciso verificar em qual das metades se deverá pesquisar o valor *e*.

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && low <= high) {  
        int mid = (low+high)/2;  
        if(v[mid]==e) found = true;  
        else if(e < v[mid])  
            ...  
        else // e > v[mid]  
            ...  
    }  
    return found;  
}
```

Se $e < v[mid]$ o elemento apenas poderá estar na primeira metade do vector: $[low]..[mid-1]$

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && low <= high) {  
        int mid = (low+high)/2;  
        if(v[mid]==e) found = true;  
        else if(e < v[mid])  
            ...  
        else // e > v[mid]  
            ...  
    }  
    return found;  
}
```

Se $e > v[mid]$ o elemento apenas poderá estar na segunda metade do vector: $[mid+1]..[high]$

Operações em Vectors

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && low <= high) {  
        int mid = (low+high)/2;  
        if(v[mid]==e) found = true;  
        else if(e < v[mid])  
            high = mid-1;  
        else // e > v[mid]  
            low = mid+1;  
    }  
    return found;  
}
```

Basta ajustar os limites da pesquisa em cada um dos casos e prosseguir...

Operações em Vectores

(Pesquisa Binária)

```
public boolean binarySearch(type e) {  
    boolean found=false;  
    int low=0, high=count-1;  
  
    while(!found && low <= high) {  
        int mid = (low+high)/2;  
        if(v[mid]==e) found = true;  
        else if(e < v[mid])  
            high = mid-1;  
        else // e > v[mid]  
            low = mid+1;  
    }  
    return found;  
}
```

Vectores de Objectos

- Até agora temos visto apenas vectores cujos elementos pertencem aos tipos elementares da linguagem: double, float, int, etc.
- Será que as mesmas operações podem ser programadas para vectores de objectos?

Vectores de Objectos

- Até agora temos visto apenas vectores cujos elementos pertencem aos tipos elementares da linguagem: `double`, `float`, `int`, etc.
- Será que as mesmas operações podem ser programadas para vectores de objectos?
 - Sim, mas é necessário alterar algumas das operações...

Vectores de Objectos

(Operadores Relacionais)

- Em algumas operações sobre vectores é necessário efectuar comparações entre dois elementos do vector.
 - Verificar se dois elementos são “iguais”...
 - Verificar se um elemento é “maior” ou “menor” do que outro...
- Mas o como definimos a igualdade entre objectos e as relações de ordem?

Vectores de Objectos

(Igualdade entre objectos)

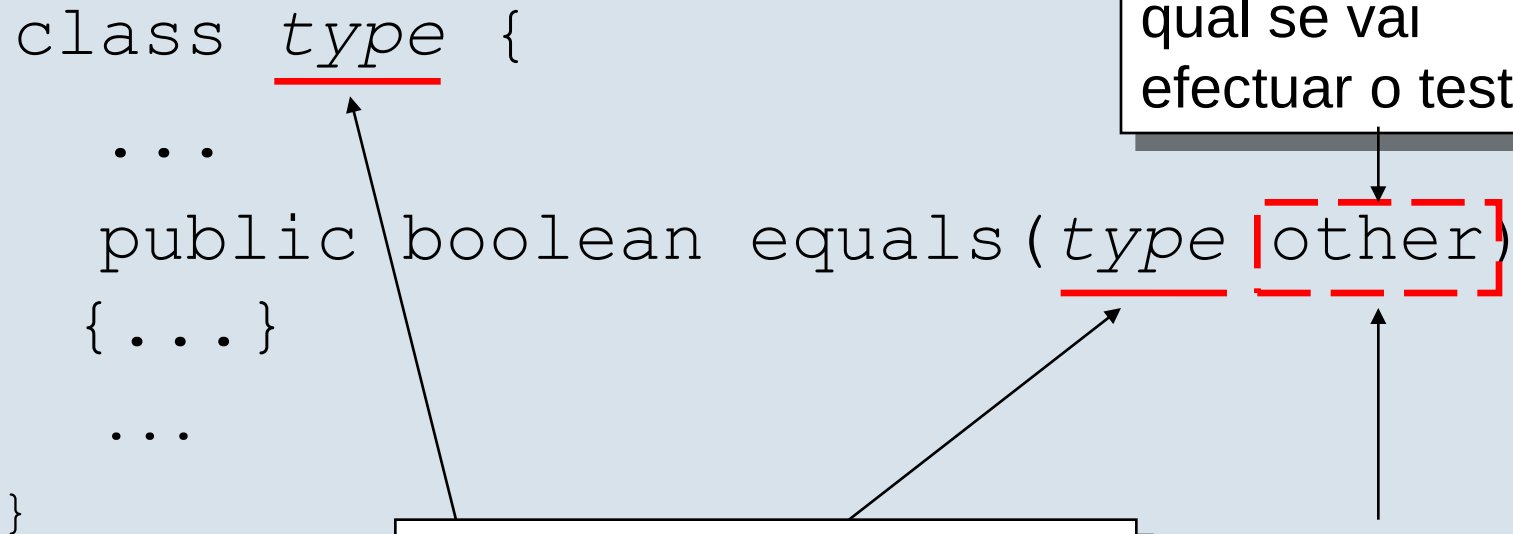
- O operador relacional `==` aplicado a objectos apenas vale **true** se de ambos os lados tivermos o mesmo objecto, o que nem sempre poderá ser o apropriado.
- No caso de dois objectos **diferentes** mas com o **mesmo estado** (o mesmo valor das suas variáveis), o operador valerá sempre **false**!

Vectores de Objectos

(Igualdade entre objectos)

- Para verificar se dois objectos da classe *type* têm o **mesmo estado** é necessário programar o método `equals`:

```
class type {  
    ...  
    public boolean equals(type other)  
    { ... }  
    ...  
}
```



O objecto com o qual se vai efectuar o teste

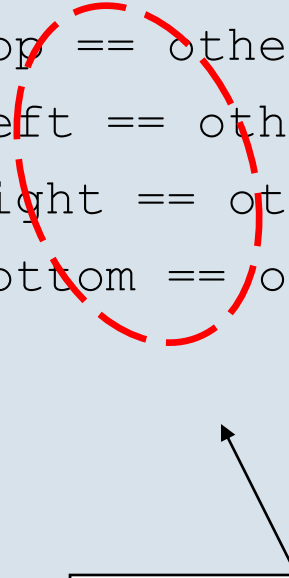
O tipo do objecto com o qual se vai comparar é, obviamente, o mesmo!

Vectores de Objectos

(Igualdade entre objectos)

- Exemplo para a primeira versão da classe Rect:

```
public boolean equals(Rect other) {  
    return  this.top == other.top &&  
           this.left == other.left &&  
           this.right == other.right &&  
           this.bottom == other.bottom;  
}
```



Como as variáveis da classe são todas de tipo elementar (double) podemos usar o operador relacional ==

Vectores de Objectos

(Igualdade entre objectos)

- Exemplo para a segunda versão da classe Rect:

```
public boolean equals(Rect other) {  
    return    topLeft.equals(other.getTopLeft()) &&  
             bottomRight.equals(other.getBottomRight());  
}
```

Neste caso, as variáveis são da classe `Point`, pelo que teremos que “confiar” no método `equals` da classe `Point`...

Vectores de Objectos

(Exemplo da Pesquisa Linear)

- Exemplo das modificações necessárias ao método de pesquisa para a frente:

```
public boolean searchForward(type e) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i].equals(e)){  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Em vez de:

$v[i] == e$

- Nos outros métodos onde se use o operador `==`, as alterações são idênticas...

Vectores de Objectos

(Exemplo da Pesquisa Linear)

- Mas, por vezes, não dispomos do objecto que pretendemos procurar na sua totalidade...
- Em vez disso, apenas dispomos de algum elemento de informação que permite identificar de forma única o objecto pretendido...
- Exemplos de cenários possíveis:
 - Procurar uma conta bancária com um determinado número de conta
 - Procurar uma conta bancária com um determinado titular
 - Procurar um determinado aluno sabendo o seu número de aluno
 - Procurar um contacto sabendo o seu e-mail
 - Procurar um contacto sabendo o seu *nickname*
 - Etc...

Vectores de Objectos

(Exemplo da Pesquisa Linear)

- Mas, por vezes, não dispomos do objecto que pretendemos procurar na sua totalidade...
- Em vez disso, apenas dispomos de algum elemento de informação (**chave**) que permite identificar de forma única o objecto pretendido...
- Neste caso podemos “dispensar” o uso do método `equals`, visto que este se aplica a dois objectos idênticos...

Vectores de Objectos

(Exemplo da Pesquisa Linear com chave)

- Exemplo das modificações necessárias ao método de pesquisa para a frente, com chave:

```
public boolean searchForward(key_type key) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i].getKey() == key) {  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Em vez de:

$v[i] == e$

Vectores de Objectos

(Exemplo da Pesquisa Linear com chave)

- Exemplo de aplicação a uma classe Student (hipotética):

```
public boolean searchForward(key_type key) {  
    boolean found = false;  
    int i=0;  
    while(i<count && !found) {  
        if(v[i].getKey() == key) {  
            found = true;  
        }  
        else i++;  
    }  
    return found;  
}
```

Em vez de:

$v[i] == e$

Vectores de Objectos

(Função de comparação)

- Em relação aos operadores relacionais ($<$, $<=$, $>$, $>=$), os mesmos não se podem aplicar a objectos, pelo que será necessário usar uma função para o efeito.
- A convenção normalmente usada na linguagem Java consiste em programar um método de comparação...

Vectores de Objectos

(Função de comparação)

- Método de comparação entre dois objectos de uma mesma classe *type*:

`(int) compareTo (type other)`

Valor a retornar pelo método:

<code>>0</code>	se <code>this</code> for considerado “maior” que <code>other</code>
<code><0</code>	se <code>this</code> for considerado “menor” que <code>other</code>
<code>0</code>	caso contrário

Vectores de Objectos

(Exemplo de ordenação)

- Exemplo das modificações necessários ao método de ordenação se aplicado a vectores de objectos:

```
public void bubbleSort() {  
    for(int i=1; i<count; i++)  
        for(int j=count-1; j>=i; j--){  
            if(v[j-1].compareTo(v[j]) > 0) {  
                type tmp = v[j-1];  
                v[j-1] = v[j];  
                v[j] = tmp;  
            }  
        }  
    }  
}
```

Vectores de Objectos

(Exemplo de Ordenação)

- Por vezes queremos ordenar por variados critérios pelo pelo que a existência de um único método de comparação não é suficiente...
- Suponhamos que queremos ordenar um vector de objectos do tipo Rect pela sua área, sem recorrer à programação explícita do método `compareTo`:

```
public void bubbleSortByArea() {  
    for(int i=1; i<count; i++)  
        for(int j=count-1; j>=i; j--){  
            if(v[j-1].getArea() > v[j].getArea()) {  
                Rect tmp = v[j-1];  
                v[j-1] = v[j];  
                v[j] = tmp;  
            }  
        }  
    }
```

Vectores

Exemplo: Classe Banco

Banco

- Associado a cada banco existe sempre um conjunto de contas bancárias. Cada conta tem um número único.
- **Operações** que são realizadas no banco:
 - Indica se existe uma dada conta, dado o número;
 - Consulta do saldo de uma dada conta, dado o número;
 - Inserção de uma nova conta, dado o número e o saldo inicial;
 - Remoção de uma dada conta, dado o número;
 - Levantamento de uma dada importância numa dada conta;
 - Depósito de uma dada importância numa dada conta;
 - Consulta da quantidade de contas com saldo inferior a zero;
 - Listagem de todas as contas do banco. Para cada conta deve ser indicado o número e saldo.

Banco

- Operações reconhecidas (interface de Bank):

public boolean hasAccount(**int** number)

Método que indica se existe a conta associada a um dado número

public int getBalance(**int** number)

Método que devolve o saldo da conta associada a um dado número

public void addAccount(**int** number, **float** amount)

Método que insere uma nova conta

public SafeAccountBank deleteAccount(**int** number)

Método que devolve e remove uma conta

public void withdraw(**int** number, **int** amount)

Método que efectua um levantamento numa dada conta

public void deposit(**int** number, **int** amount)

Método que efectua um depósito numa dada conta

public int getRedZone()

Método que indica o número de contas com saldo negativo

public SafeBankAccount getAccount(**int** number)

Método que devolve a conta associada a um dado número

Banco

- Operações reconhecidas (interface de Bank):
 - Para listar a informação referente a todas as contas vamos ter várias operações para iterar entre todas as contas. Assumimos que durante o percurso pelas contas não são feitas inserções nem remoções de contas.

```
public void initAccountList()
```

Método que inicia a listagem das contas do banco

```
public boolean hasNextAccount()
```

Método que indica se existe uma conta seguinte

```
public String nextAccount()
```

Método que devolve a informação referente à conta seguinte, em String

Conta Bancária Segura

- Associado a cada conta bancária existe sempre um número. Este número identifica a conta bancária, ou seja, não existem duas contas com o mesmo número no banco.
- Queremos que a classe apresente mais duas novas operações na sua interface:

```
public int getNumber()
```

Consulta o número da conta

```
public String toString()
```

Devolve a descrição da conta (número e saldo)

Classe SafeBankAccount

```
public class SafeBankAccount () {
```

```
...
```

```
private int number;
```

Nova variável de instância

```
public SafeBankAccount(int n) {
```

```
...
```

```
}
```

Novo Parâmetro

```
public SafeBankAccount(int n, int amount) {
```

```
...
```

```
}
```

```
...
```

Os construtores devem ser alterados para obrigar a que uma conta bancária tenha sempre um número

```
}
```

Banco

- Associado a cada banco existe sempre um conjunto de contas bancárias. Cada conta tem um número único.
 - Necessitamos poder guardar n contas bancárias, logo é necessário poder ter n variáveis da classe (tipo) SafeBankAccount.
 - Assim, vamos precisar de:
 - Vector de contas bancárias com uma dada capacidade (máxima);
 - Número que me indica sempre quantas contas existem no vector (contador de contas em utilização).

Classe Bank

```
public class Bank {  
    private SafeBankAccount[] accounts;  
  
    private int countAccounts;  
  
    private int currentAccount;  
    public Bank() {  
        ...  
    }  
    ...  
}
```

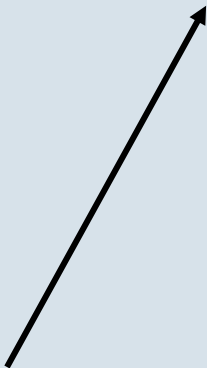
Vector de contas

Número de contas existentes
no vector

Posição no vector da
próxima conta a mostrar,
quando se está a listar
todas as contas

Classe Bank

```
public class Bank {  
    private static final int MAXACCOUNTS = 1000;  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    private int currentAccount; // posição da conta corrente  
  
    public Bank() {  
        accounts = new SafeBankAccount[MAXACCOUNTS];  
        countAccounts = 0;  
        currentAccount = -1;  
    }  
    ...  
}
```



Vector de, no máximo, 1000 contas

Classe Bank

```
public class Bank {  
    private static final int MAXACCOUNTS =1000;  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    private int currentAccount; // posição da conta corrente  
  
    public boolean hasAccount(int number){  
        int position = findAccountPosition(number);  
        return position != -1;  
    }  
    ...  
}
```



**Método que retorna a posição no
vector onde está a conta com o
número dado**

Classe Bank

Pesquisa dum elemento num vector:

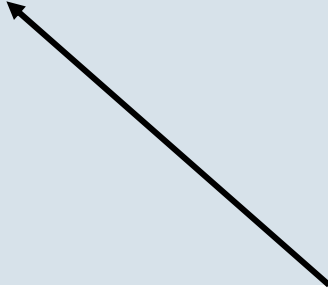
–Percorre os elementos do vector até encontrar o elemento

```
public class Bank {  
    ...  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    ...  
    private int findAccountPosition(int number) {  
        int i = 0; // percurso pelos elementos 0..countAccounts  
        boolean find = false; // indicador de existência  
        while((i < countAccounts) && (!find))  
            if( accounts[i].getNumber() == number)  
                find = true;  
            else i++;  
        if (find) return i;  
        else return -1;  
    }  
    ...  
}
```

**Objecto SafeBankAccount guardado
na posição i do vector**

Classe Bank

```
public class Bank {  
    private static final int MAXACCOUNTS =1000;  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    ...  
    /* Pré-condição: a conta existe */  
    public int getBalance(int number){  
        int position = findAccountPosition(number);  
        return accounts[position].getBalance();  
    }  
    ...  
}
```



Objecto SafeBankAccount pretendido

Classe Bank

```
public class Bank{  
    static private final int MAXACCOUNTS =1000;  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    ...  
    public void addAccount(int number, int amount){  
        if ((countAccounts < MAXACCOUNTS) &&  
            !hasAccount(number)){  
            accounts[countAccounts]=  
                new SafeBankAccount(number,amount);  
            countAccounts++;  
        }  
    }  
    ...  
}
```

Colocação na primeira posição vazia do vector

Criação do novo objecto SafeBankAccount

Actualização do número de elementos no vector

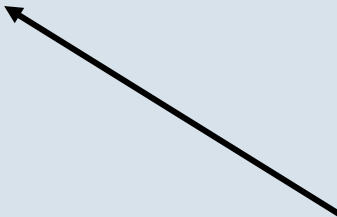
Classe Bank

```
public class Bank {  
    ...  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    ...  
    public SafeBankAccount removeAccount(int number){  
        SafeAccountBank el= null;  
        int position = findAccountPosition(number);  
        if (position != -1){  
            el = accounts[position];  
            for (int i = position; i<countAccounts-1; i++)  
                accounts[i] = accounts[i+1];  
            countAccounts--;  
        }  
        return el;  
    }  
    ...  
}
```

**Objecto SafeBankAccount
pretendido**



**O vector não deve ficar com elementos vazios.
Logo temos que mover todos os elementos desde
a posição do elemento a remover até ao final**

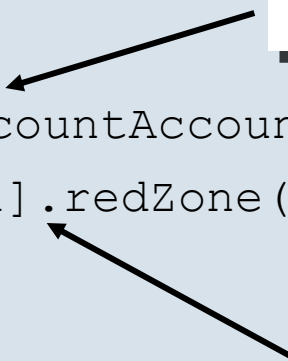


Classe Bank

```
public class Bank {  
    ...  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    ...  
    public void withdraw(int number, int amount){  
        int position = findAccountPosition(number);  
        if (position != -1)  
            accounts[position].withdraw(amount);  
    }  
    public void deposit(int number, int amount){  
        int position = findAccountPosition (number);  
        if (position != -1)  
            accounts[position].deposit(amount);  
    }  
    ...  
}
```

Classe Bank

```
public class Bank {  
    ...  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    ...  
    public int getRedZone() {  
        int count = 0;  
        for (int i = 0; i < countAccounts; i++)  
            if (accounts[i].redZone())  
                count++;  
        return count;  
    }  
    ...  
}
```



Percurso por todos os elementos do vector

Para cada elemento que satisfaz o critério (ter saldo negativo) deve-se incrementar o contador

Banco

- Operações reconhecidas (interface de Bank):
 - Para listar a informação referente a todas as contas vamos ter várias operações para iterar entre todas as contas. Assumimos que durante o percurso pelas contas não são feitas inserções nem remoções de contas.

```
public void initAccountList()
```

Método que inicia a listagem das contas no banco

```
public boolean hasNextAccount()
```

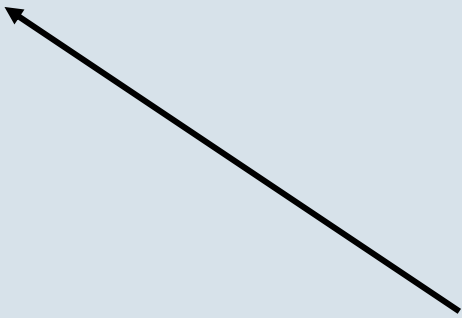
Método que indica se existe seguinte conta

```
public String nextAccount()
```

Método que devolve a informação referente à seguinte conta

Classe Bank

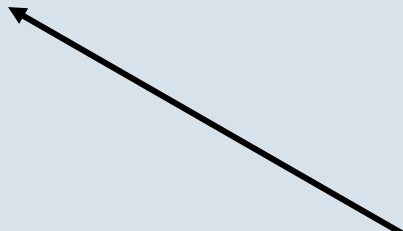
```
public class Bank {  
    static private final int MAXACCOUNTS =1000;  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    private int currentAccount; // posição da conta corrente  
  
    public void initAccountList() {  
        if (countAccount != 0)  
            currentAccount = 0;  
        else currentAccount = -1  
    }  
    ...  
}
```



**Coloca a conta corrente da iteração
como sendo a conta que está na
posição 0**

Classe Bank

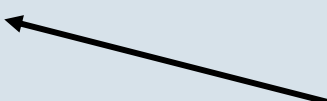
```
public class Bank {  
    static private final int MAXACCOUNTS =1000;  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    private int currentAccount; // posição da conta corrente  
  
    public boolean hasNextAccount() {  
        return ((currentAccount >= 0 ) &&  
            (currentAccount < countAccounts));  
    }  
    ...  
}
```



Só existem mais contas para listar se a posição da conta corrente for menor que o número de elementos e maior ou igual a 0

Classe Bank

```
public class Bank {  
    ...  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    private int currentAccount; // posição da conta corrente  
  
    public String nextAccount() {  
        String account = null;  
        if ( hasNextAccount() ) {  
            account = accounts[currentAccount].toString();  
            currentAccount ++;  
        }  
        return account;  
    }  
    ...  
}
```



Devolve a descrição da conta e prepara a conta corrente para a próxima iteração (nextAccount)

Classe MainBank

```
public class MainBank () {  
    public static void main(String[] args) {  
        Bank myBank = new Bank();  
        myBank.addAccount(12345, 10000);  
        myBank.addAccount(22345, 20000);  
        myBank.addAccount(32345, 30000);  
        myBank.addAccount(42345, 0);  
  
        Listagem das contas do banco  
        myBank.initAccountList();  
        while ( myBank.hasNextAccount() )  
            System.out.println(myBank.nextAccount());  
    }  
}
```



Banco

- Para poder ordenar as contas bancárias do banco é necessário saber ordenar duas contas.
 - Como saber se uma conta é menor, maior ou igual a outra conta?
 - Ordenação por número de conta
- Logo necessitamos que a classe `SafeBankAccount` apresente a seguinte nova operação na sua interface:

`int compareTo(SafeBankAccount c)`

Retorna 0 caso as contas tenham o mesmo número, 1 se a conta corrente (this) tiver um número de conta superior à conta c e -1, caso contrário.

Classe SafeBankAccount

```
public class SafeBankAccount{  
    ...  
  
    public int compareTo(SafeBankAccount c) {  
        if ( number == c.getNumber())  
            return 0;  
        if ( number > c.getNumber())  
            return 1;  
        return -1;  
    }  
    ...  
}
```

Classe Bank

```
public class Bank {  
    ...  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
  
    /* Ordenação por ordem crescente de número */  
    public void orderByNumber() {  
        int i, j;  
        SafeBankAccount aux;  
        for (i = 1; i < countAccounts; i++)  
            for (j = countAccounts - 1; j >= i; j--)  
                if (accounts[j-1].compareTo(accounts[j]) > 0) {  
                    aux = accounts[j-1];  
                    accounts[j-1] = accounts[j];  
                    accounts[j] = aux;  
                }  
    }  
    ...  
}
```

Pesquisa Binária no Banco

```
public int binarySearchAccountPosition(int number) {
    int mid;
    int low = 0;
    int high = countAccounts - 1;

    while (low <= high) {
        mid = (low + high) / 2;
        if (accounts[mid].getNumber() == number)
            return mid;
        else if (accounts[mid].getNumber() < number)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
```

Conta Bancária

- Associado a cada conta bancária existe sempre um titular.
- Queremos que a classe apresente a seguinte nova operação na sua interface:

`String getHolder()`

Indica o nome do titular da conta

Classe SafeBankAccount

```
public class SafeBankAccount {
```

```
...
```

Nova variável de instância



```
private String holderName;
```

```
private SafeBankAccount(int number, String name) {
```

```
...
```

```
}
```

Novo Parâmetro



```
private SafeBankAccount(int number, String name,  
                        int amount) {
```

```
...
```

```
}
```

```
...
```

**Os construtores devem ser alterados para
obrigar a que uma conta bancária tenha
sempre um titular**

```
}
```


Banco

- Necessitamos deste modo alterar a operação `addAccount`, de modo a receber um novo parâmetro que é o nome do titular da conta.

```
void addAccount(int number, String name, float amount)
```

- É necessário apresentar as seguintes novas operações na sua interface:

```
int findAccountNumber(String name)
```

Retorna o número de uma conta cujo titular seja o nome dado e -1, caso não exista conta com esse titular.

```
void orderByHolder()
```

Ordena as contas por nome de titular

Classe Bank

```
public class Bank {  
    private static final int MAXACCOUNTS =1000;  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    private int currentAccount; // posição da conta corrente  
  
    public int findAccountNumber(String name){  
        int position = findAccountPosition(name);  
        if (position != -1)  
            return accounts[position].getNumber();  
        return -1;  
    }  
    ...  
}
```



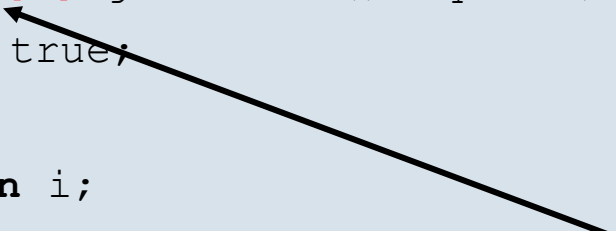
Método que retorna a posição no vector onde está a primeira conta do titular dado

Classe Bank

Pesquisa dum elemento num vector:

–Percorre os elementos do vector até encontrar o elemento

```
public class Bank {  
    ...  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    ...  
    private int findAccountPosition(String name){  
        int i = 0; // percurso pelos elementos 0..countAccounts  
        boolean find = false; // indicador de existência  
        while((i < countAccounts) && (!find))  
            if( accounts[i].getHolder().equals(name))  
                find = true;  
            else i++;  
        if (find) return i;  
        else return -1;  
    }  
    ...  
}
```



**Objecto SafeBankAccount guardado
na posição i do vector**

Classe Bank

```
public class Bank {  
    ...  
    private SafeBankAccount[] accounts; // vector de contas  
    private int countAccounts; // número de contas no vector  
    /* Ordenação por ordem crescente de nome de titular */  
    public void orderByHolder() {  
        int i, j;  
        SafeBankAccount aux;  
        for (i = 1; i < countAccounts; i++)  
            for (j = countAccounts - 1; j >= i; j--)  
                if (accounts[j-1].getHolder().  
                    compareTo(accounts[j].getHolder()) > 0) {  
                    aux = accounts[j-1];  
                    accounts[j-1] = accounts[j];  
                    accounts[j] = aux;  
                }  
    }  
    ...  
}
```