Vectores - Pesquisa Linear

Mestrado Integrado em Engenharia Informática FCT UNL
http://ctp.di.fct.unl.pt/miei/ip/

Corpo Docente 2020/2021
António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

Operações em Vectores

Revisão (para estudo autónomo)

Operações em Vectores

Classe de exemplo

```
public class MyVector {
  private static final int DEFAULT_CAPACITY = ???;
  private static final int GROWTH = ???; // maior que 1
  private type[] v;
  private int count;
  public MyVector() {
    v = new type[DEFAULT_CAPACITY];
    count = 0;
  }
  public int size();
  public void insertLast(type e);
  public void removeLast();
  public void insertAt(type e, int pos);
  public void removeAt(int pos);
  public boolean searchForward(type e);
  public boolean searchBackward(type e);
}
```

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

Operações em Vectores

Classe de exemplo

```
public class MyVector {
  private static final int DEFAULT_CAPACITY = ???;
  private static final int GROWTH = ???; // maior que 1
  private type[] v;
  private int count;
  public MyVector() {
    v = new type[DEFAULT_CAPACITY];
    count = 0;
  }
  public int size();
  public void insertLast(type e);
  public void removeLast();
  public void insertAt(type e, int pos);
  public void removeAt(int pos);
  public boolean searchForward(type e);
  public boolean searchBackward(type e);
}
```

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

Operações em Vectores

Classe de exemplo para o estudo das operações

```
public class MyVector {
  private static final int DEFAULT_CAPACITY = ???;
  private static final int GROWTH = ???; // maior que 1
  private type[] v;
  private int count;
  public MyVector() {
    v = new type[DEFAULT_CAPACITY];
    count = 0;
  }
  public int size();
  public void insertLast(type e);
  public void removeLast();
  public void insertAt(type e, int pos);
  public void removeAt(int pos);
  public boolean searchForward(type e);
  public boolean searchBackward(type e);
}
```

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

Operações em Vectores

Classe de exemplo para o estudo das operações

```
public class MyVector {
  private static final int DEFAULT_CAPACITY=???;
  private static final int GROWTH = ???; // maior que 1
  private type[] v;
  private int count;
  public MyVector();
  public int size() {
    return count;
  }
  public void insertLast(type e);
  public void removeLast();
  public void insertAt(type e, int pos);
  public void removeAt(int pos);
  public boolean searchForward(type e);
  public boolean searchBackward(type e);
}
```

Precisamos também de saber quantos elementos tem a colecção em cada momento. Continuamos a usar `count`, pois na maioria dos casos, só uma parte do vector está ocupada.

Operações em Vectores

Classe de exemplo para o estudo das operações

```
public class MyVector {
  private static final int DEFAULT_CAPACITY=???;
  private static final int GROWTH = ???; // maior que 1
  private type[] v;
  private int count;
  public MyVector();
  private boolean isFull();
  private void resize();
  public int size();
  public void insertLast(type e);
  public void removeLast();
  public void insertAt(type e, int pos);
  public void removeAt(int pos);
  public boolean searchForward(type e);
  public boolean searchBackward(type e);
}
```

Tal com anteriormente, esta classe não tem um limite explícito na sua capacidade. Assim sendo, a gestão interna do vector continua a ser feita por meio de métodos **privados**.

Operações em Vectores

Classe de exemplo para o estudo das operações

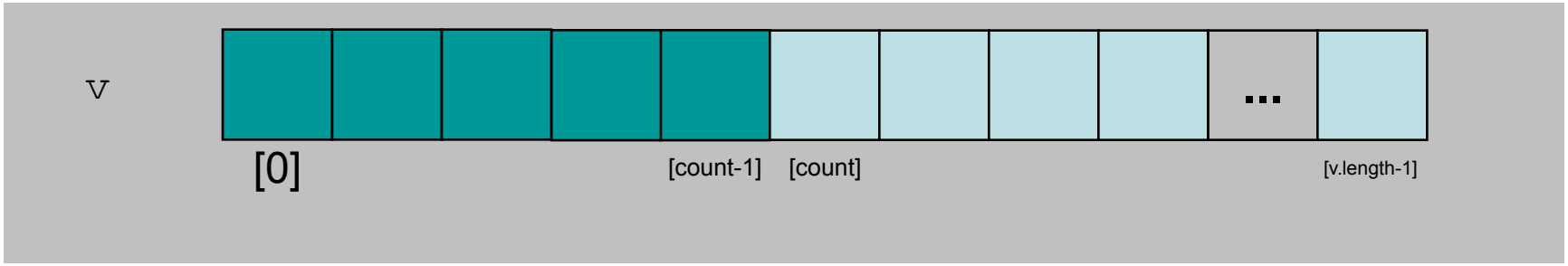
```
public class MyVector {
  private static final int DEFAULT_CAPACITY = ???;
  private static final int GROWTH = ???; // maior que 1
  private type[] v;
  private int count;
  public MyVector();

  private boolean isFull() {
    return count == v.length;
  }
  private void resize() {
    type[] tmp = new type[GROWTH * v.length];
    for ( int i = 0; i < v.length; i++ )
       tmp[i] = v[i];
    v = tmp;
  }
  ...
```

As implementações de `isFull()` e `resize()` são essencialmente as mesmas que anteriormente.

Operações em Vectores
Inserção no final

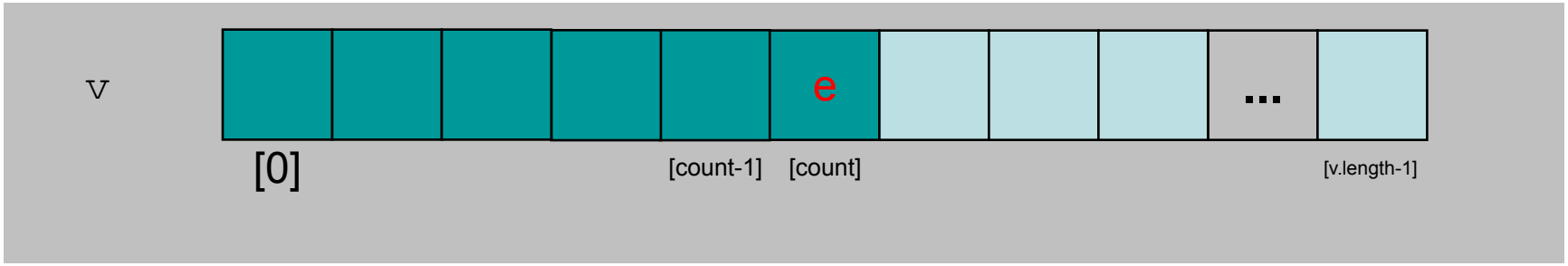
```
public void insertLast(type e) {
    if (isFull())
        resize();
    v[count] = e;
    count++;
}
```



Operações em Vectores

Inserção no final

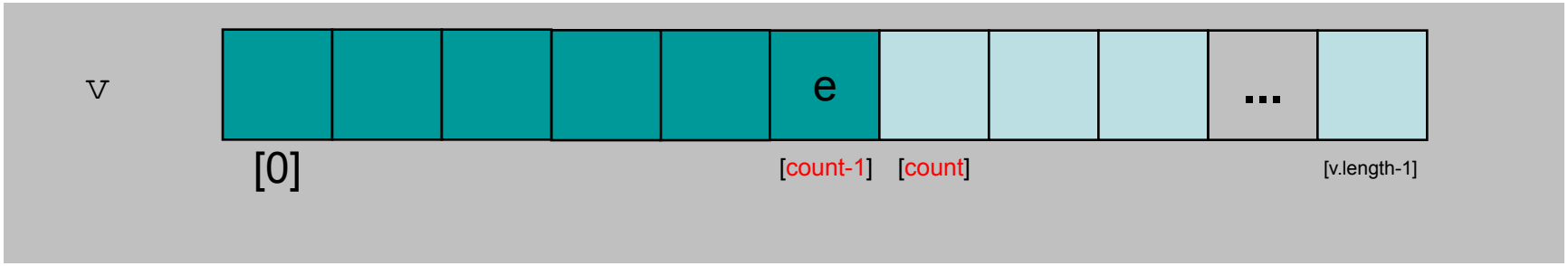
```
public void insertLast(type e) {
    if (isFull())
        resize();
    v[count] = e;
    count++;
}
```



Operações em Vectores

Inserção no final

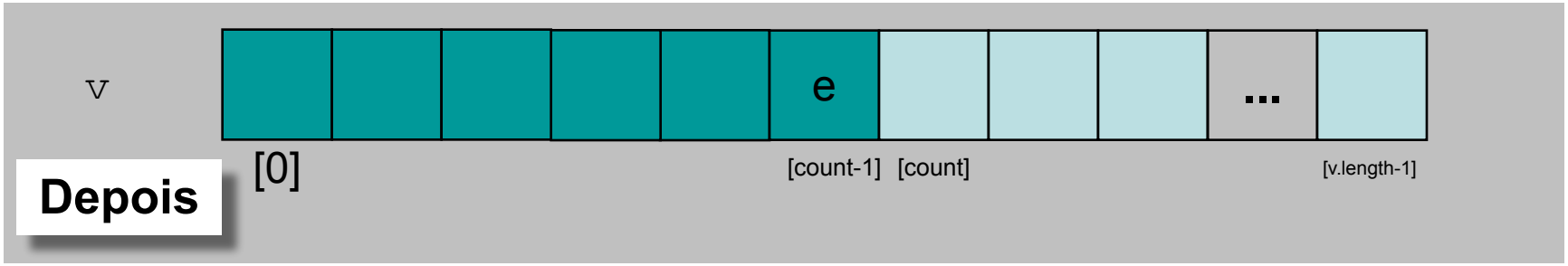
```
public void insertLast(type e) {
    if (isFull())
        resize();
    v[count] = e;
    count++;
}
```

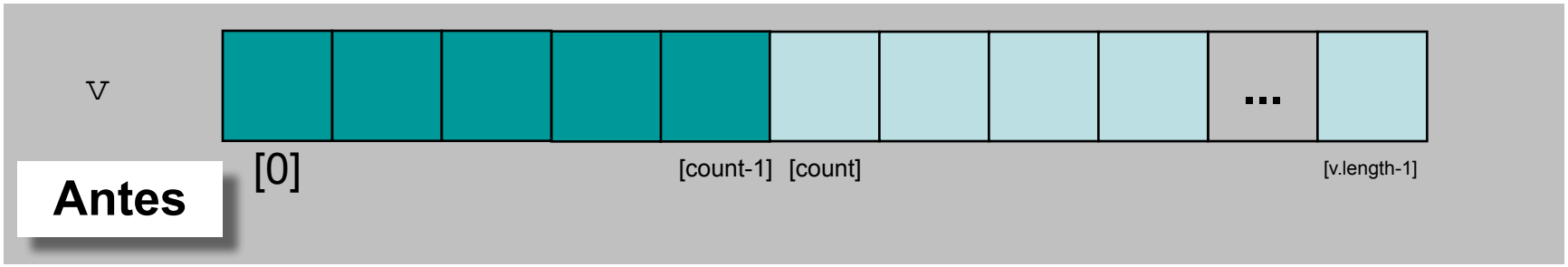


Operações em Vectores

Inserção no final

```
public void insertLast(type e) {
    if (isFull())
        resize();
    v[count] = e;
    count++;
}
```



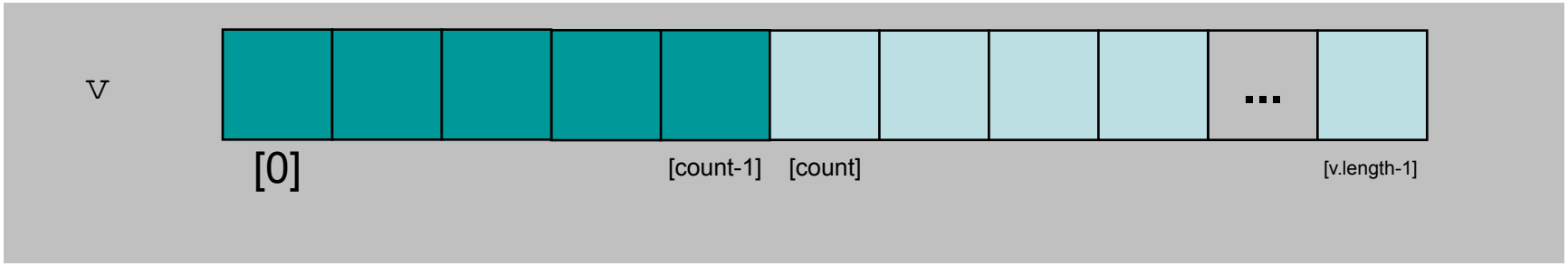


Operações em Vectores

Remoção do final

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

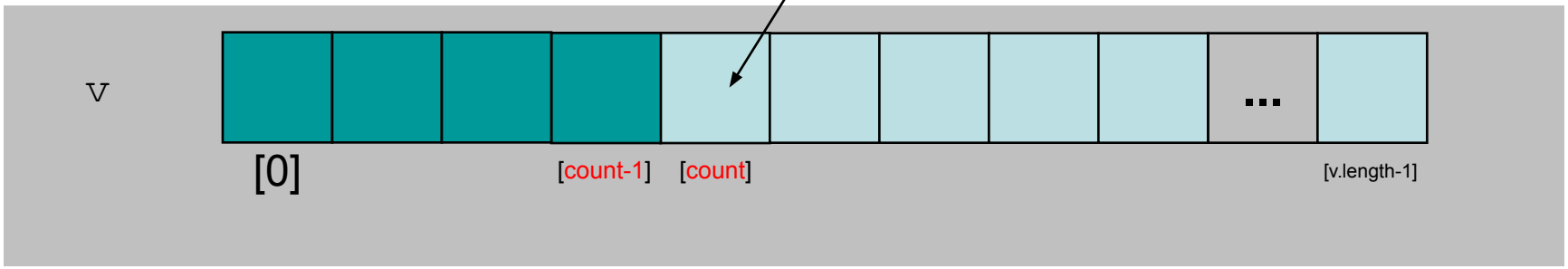
O vector tem que ter pelo menos um elemento.



Operações em Vectores
Remoção do final

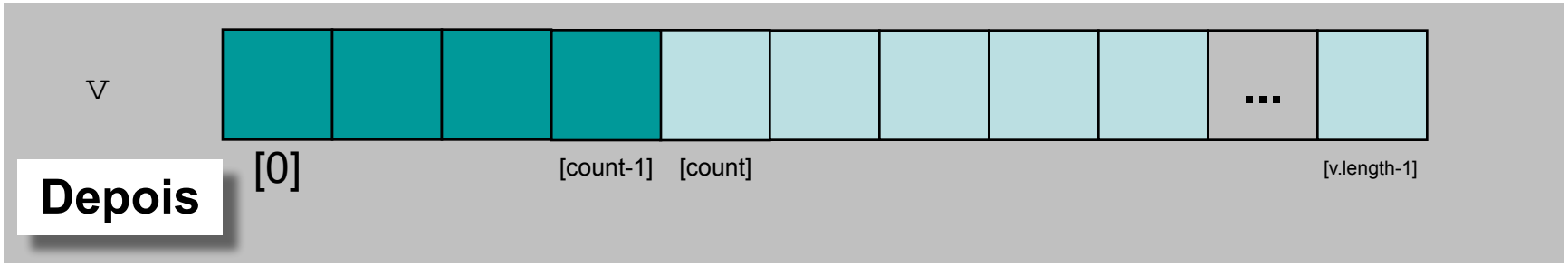
```
/** pre: size() > 0 */
public void removeLast() {
   count--;
}
```

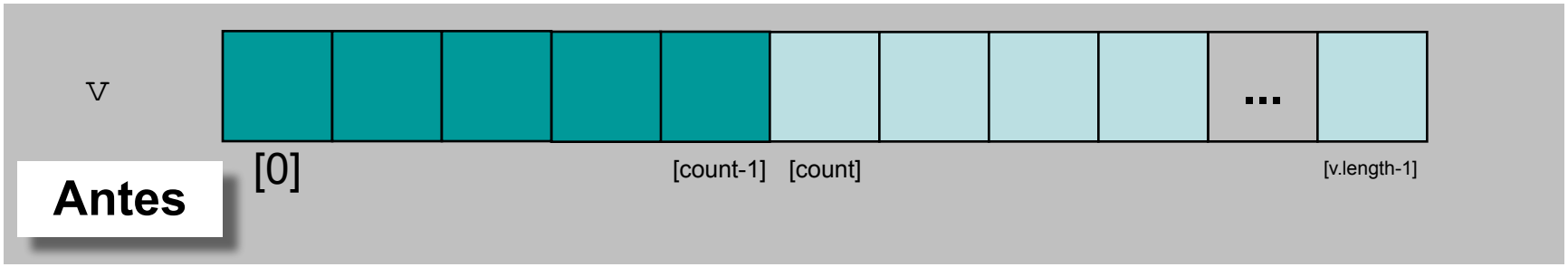
Basta decrementar o contador de elementos! O que antes era o último elemento permanece no vector mas é “esquecido” em futuras operações.



Operações em Vectores
Remoção do final

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



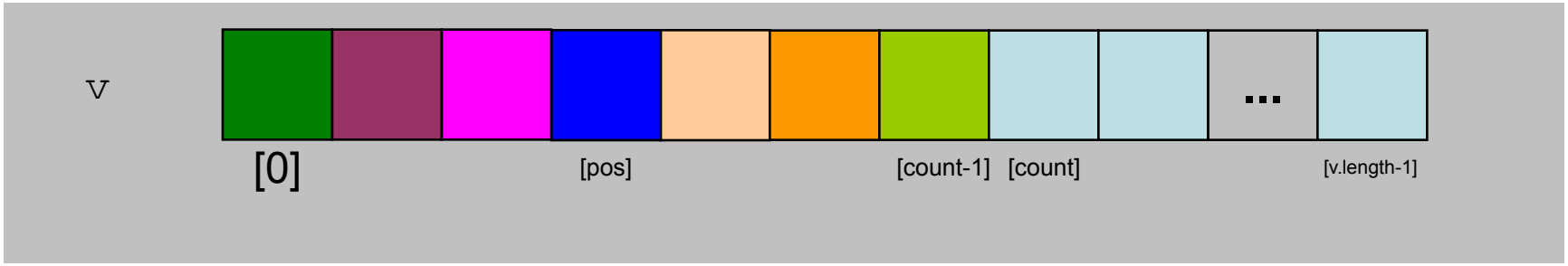


Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

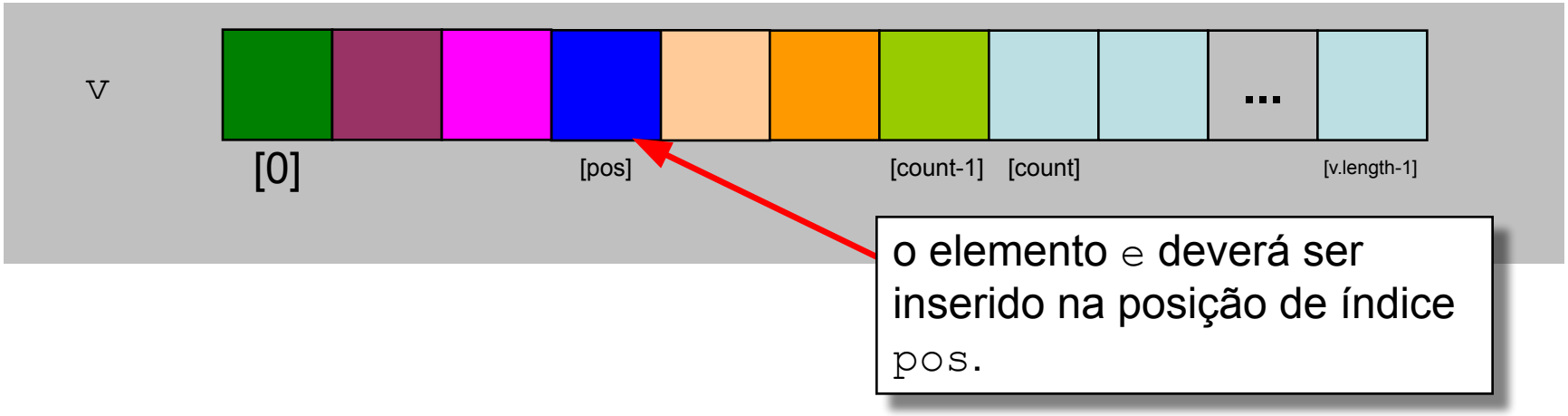
A posição dada tem de estar preenchida ou ser a primeira vazia



Operações em Vectores

Inserção preservando a ordem dos existentes

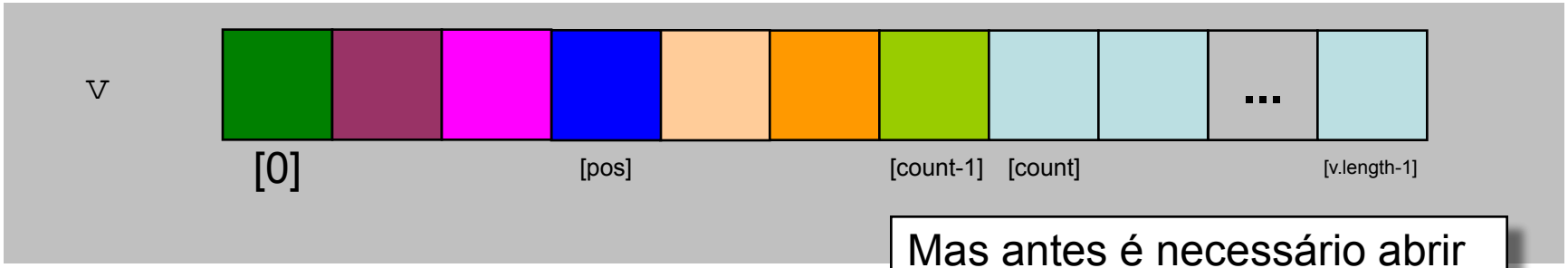
```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```



Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

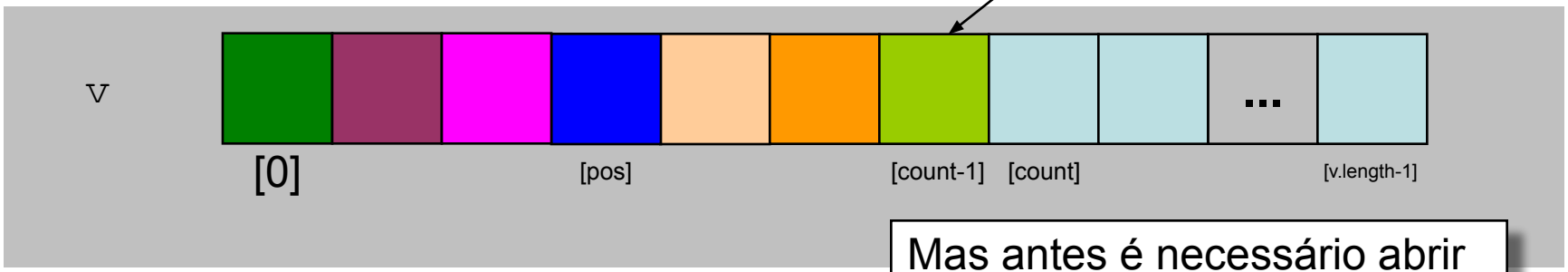


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

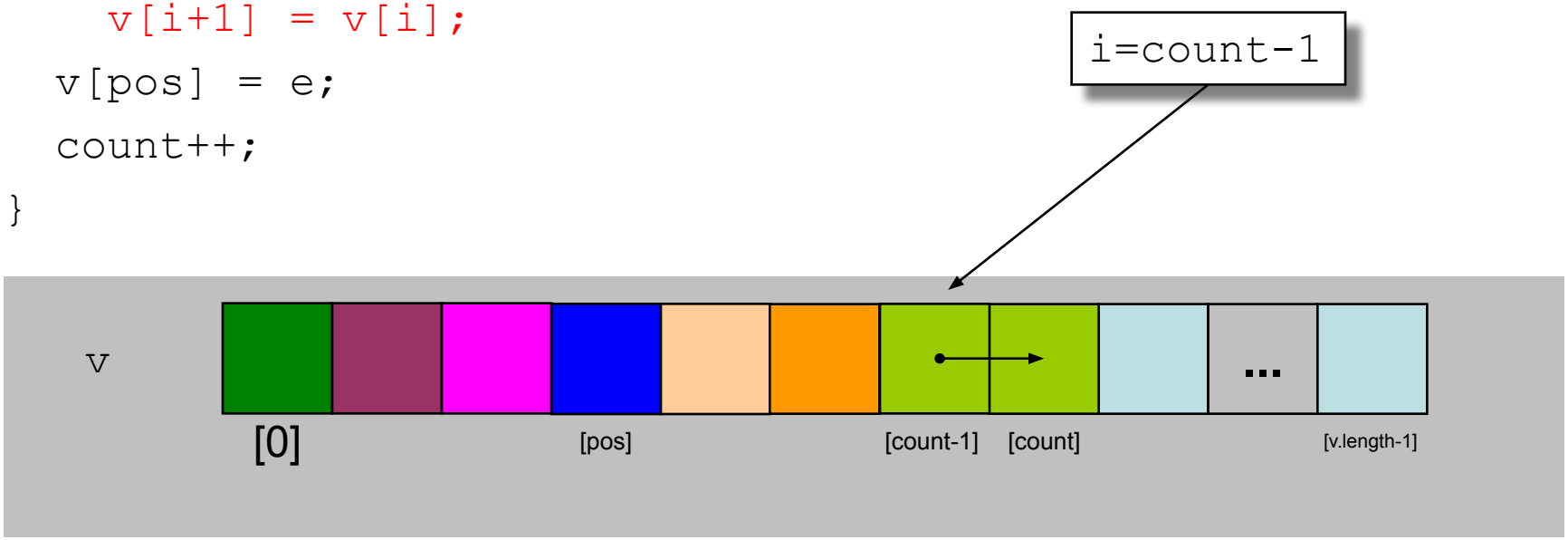


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

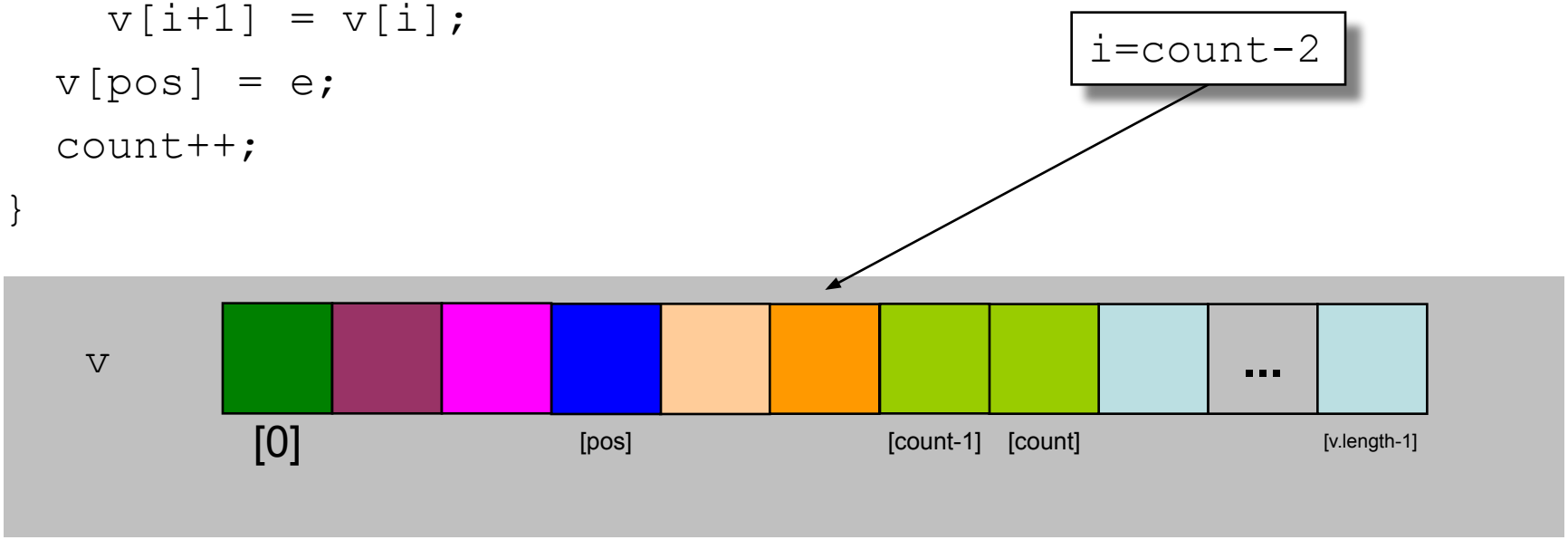


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

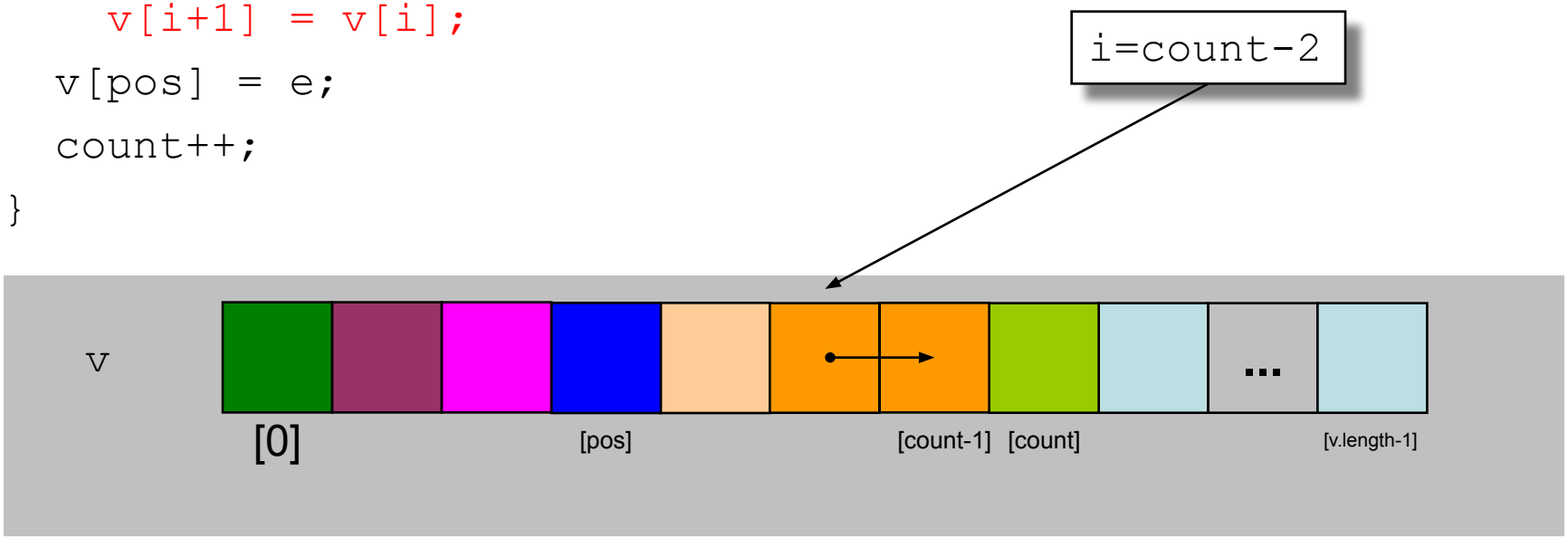


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

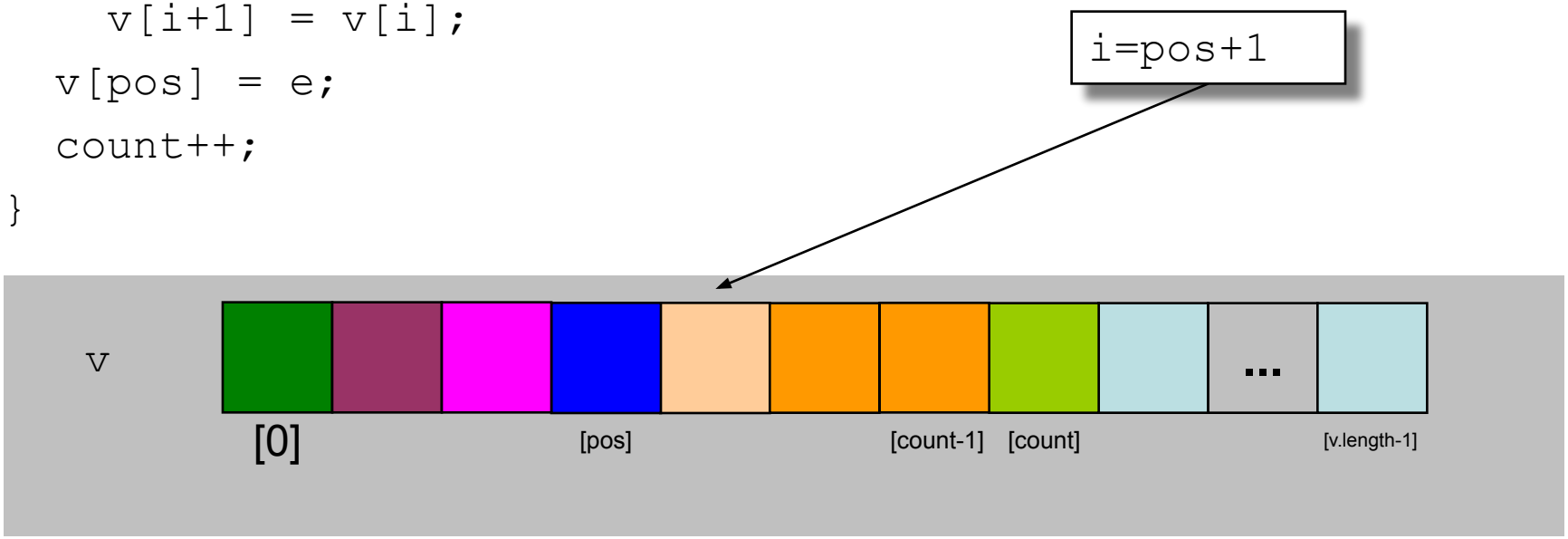


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

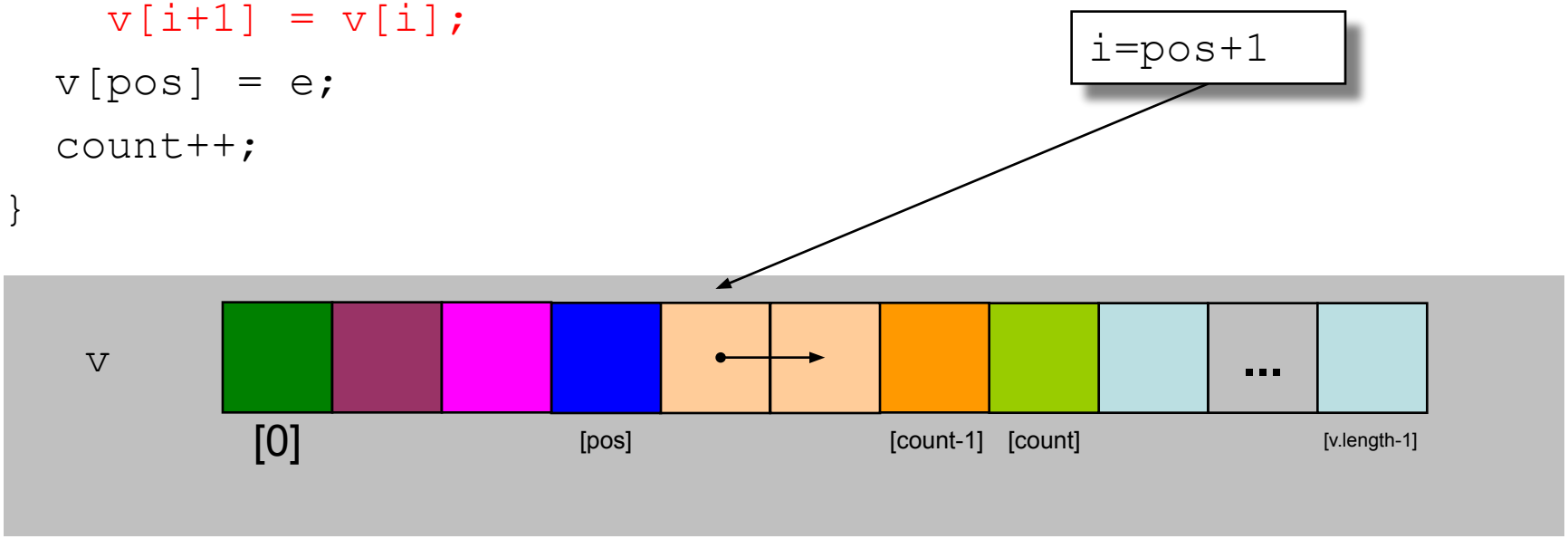


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

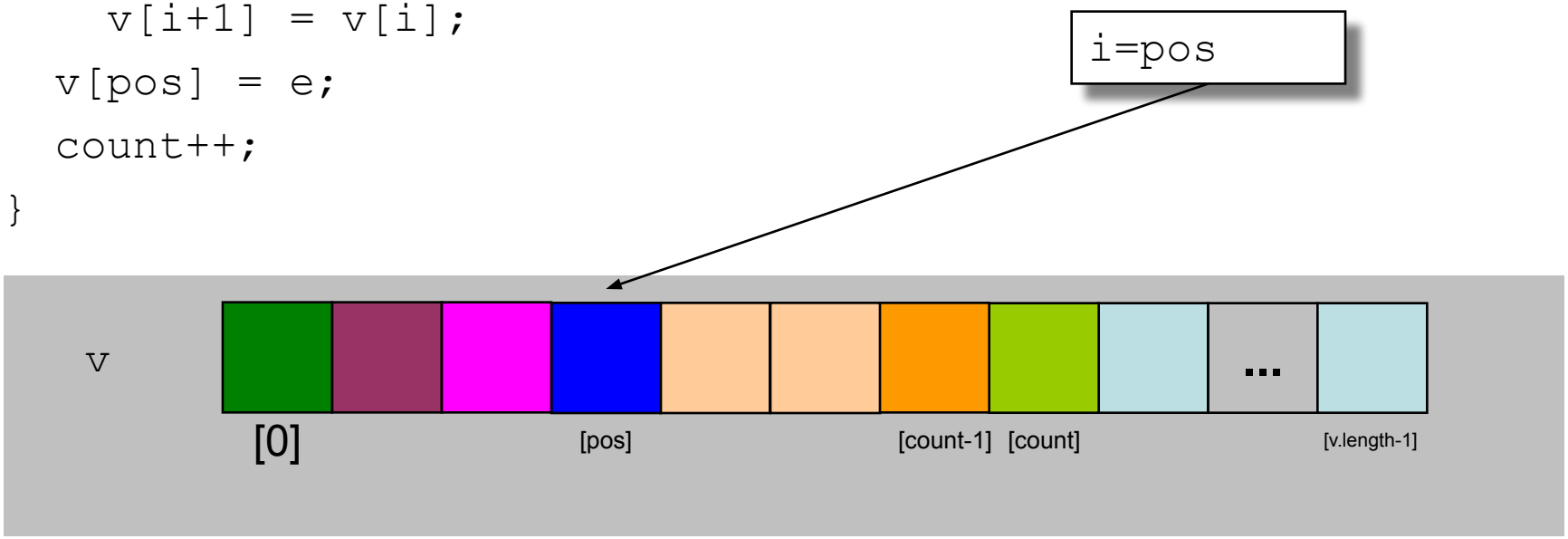


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

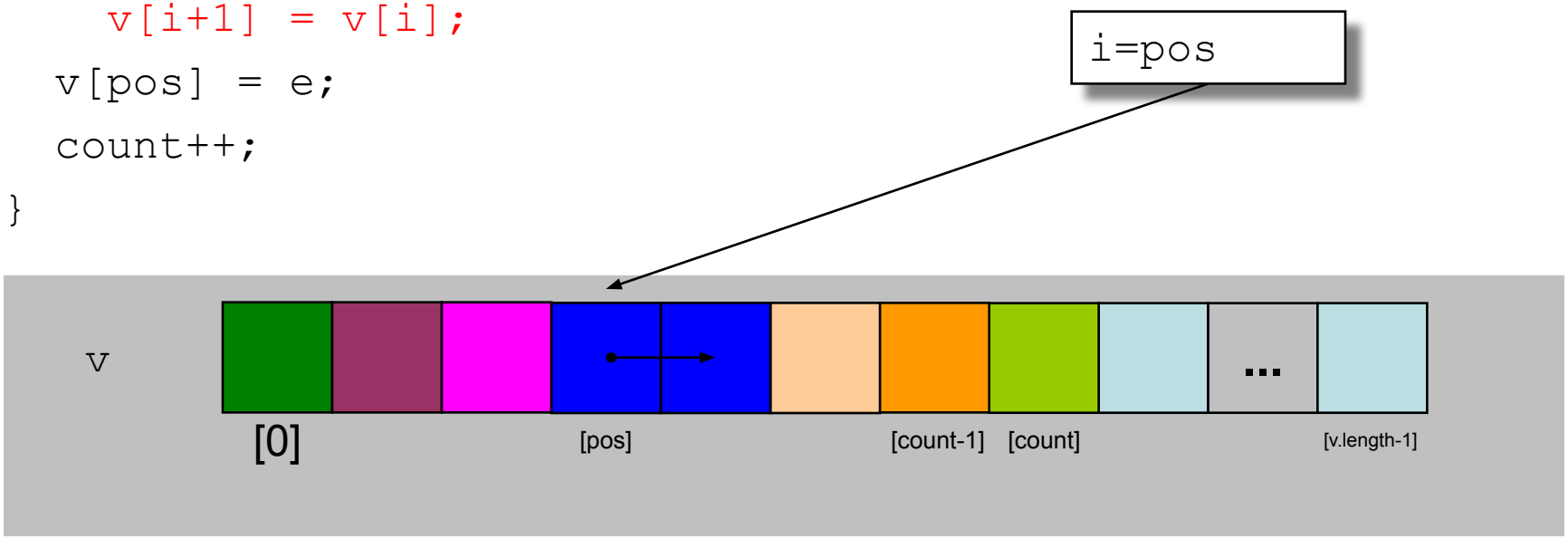


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

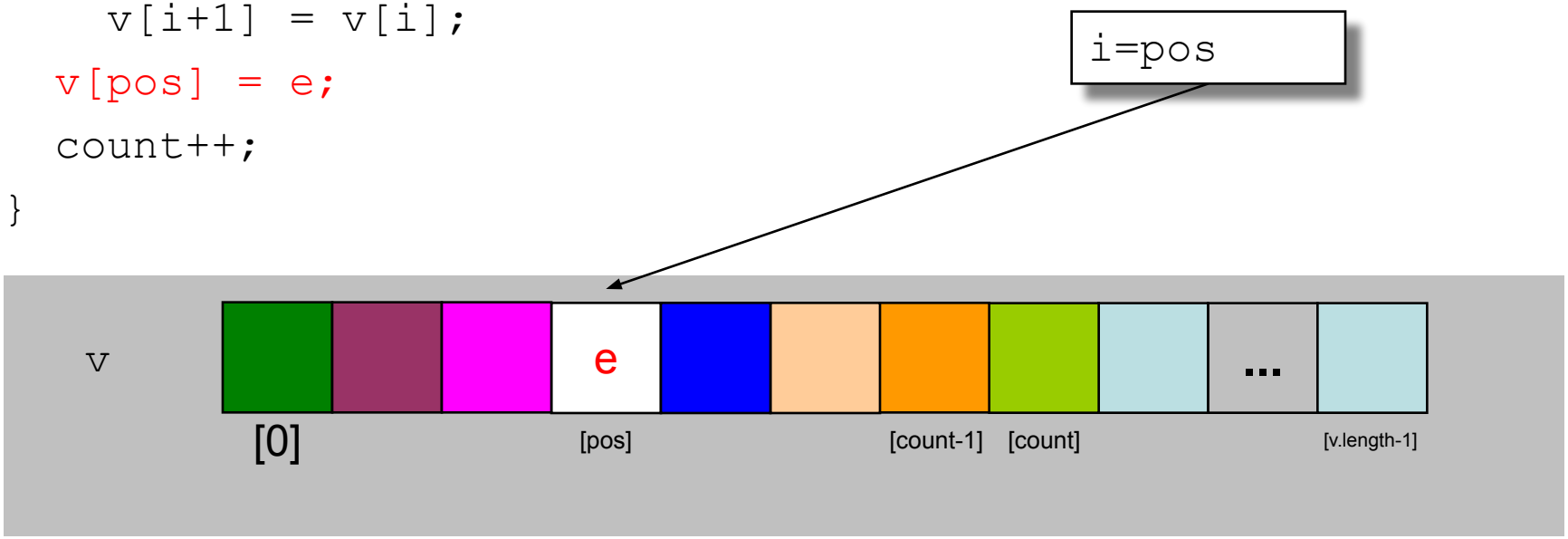


Mas antes é necessário abrir espaço para o lá colocar...

Operações em Vectores

Inserção preservando a ordem dos existentes

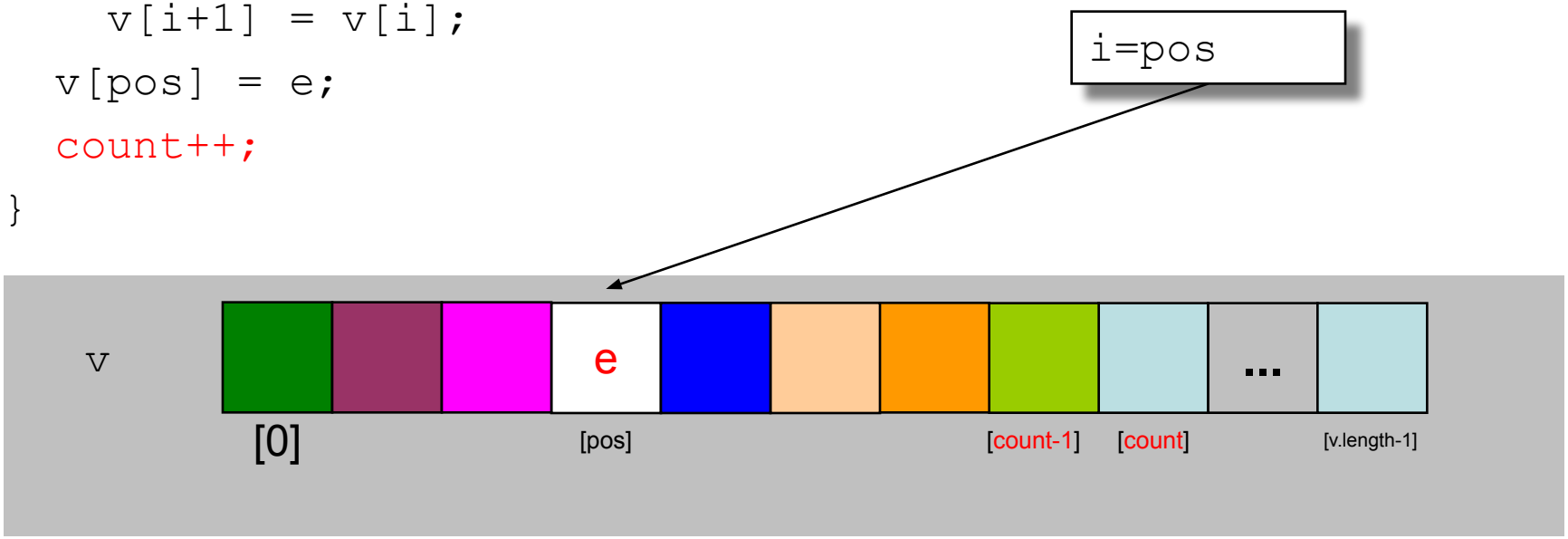
```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```



Operações em Vectores

Inserção preservando a ordem dos existentes

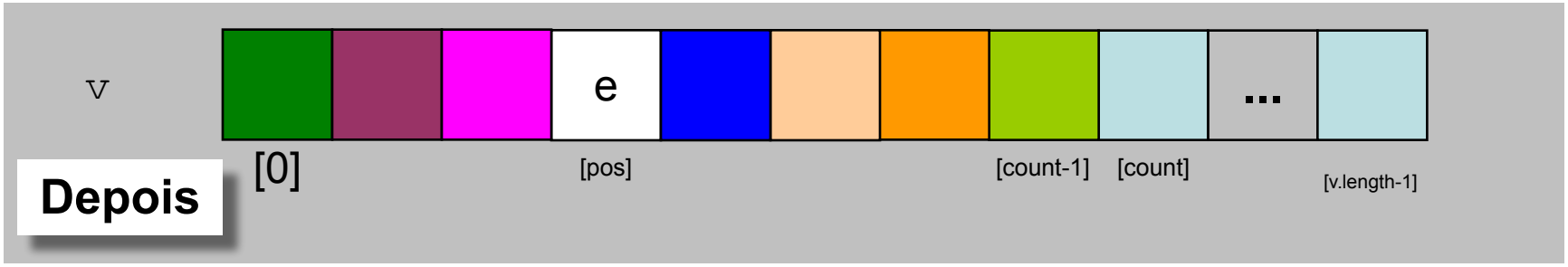
```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```

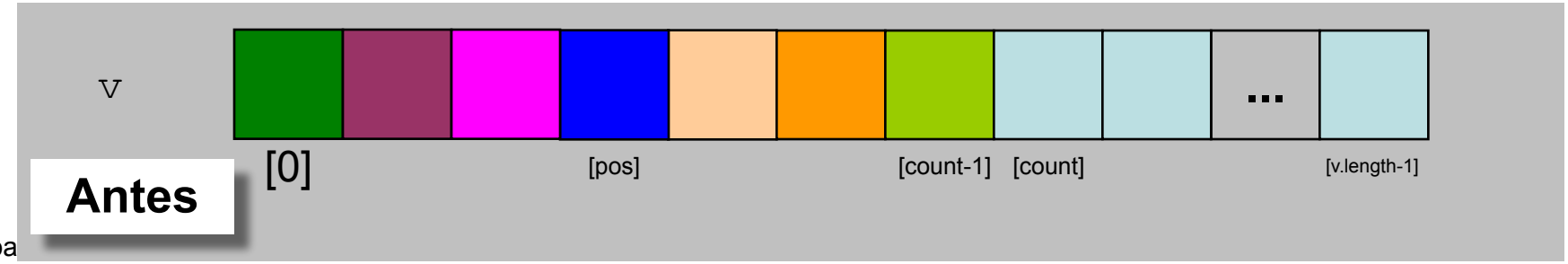


Operações em Vectores

Inserção preservando a ordem dos existentes

```
/** 0 <= pos && pos <= size() */
public void insertAt(type e, int pos) {
  if (isFull()) resize();
  for (int i=count-1; i >= pos; i--)
    v[i+1] = v[i];
  v[pos] = e;
  count++;
}
```



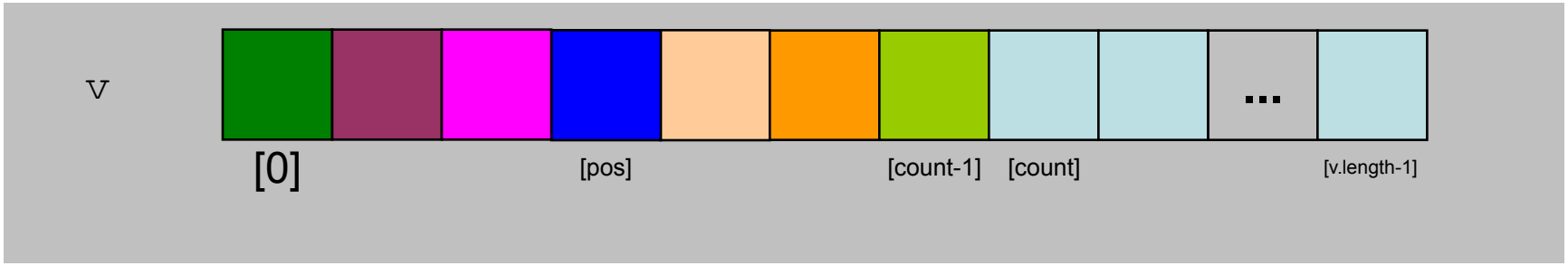


Operações em Vectores

Remoção preservando a ordem dos que ficam

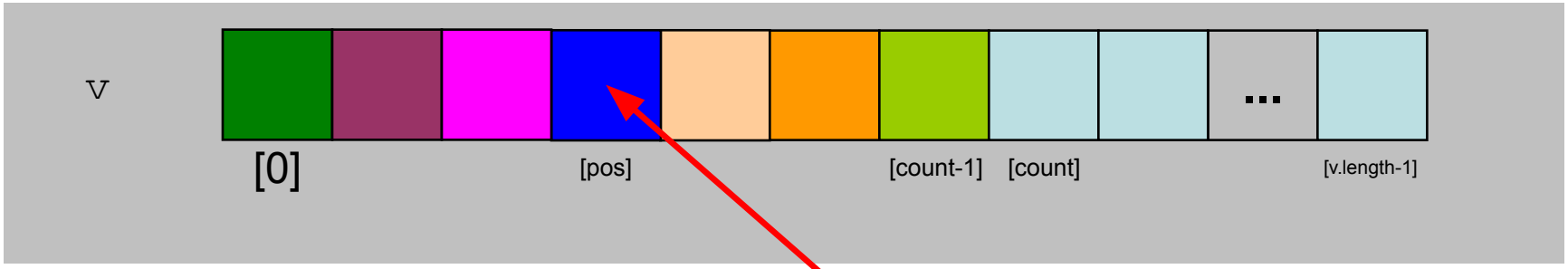
```
/** pre: 0 <= pos && pos < size() */
public void removeAt(int pos) {
    for (int i=pos; i<count-1; i++)
        v[i] = v[i+1];
    count--;
}
```

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



Operações em Vectores
Remoção preservando a ordem dos que ficam

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

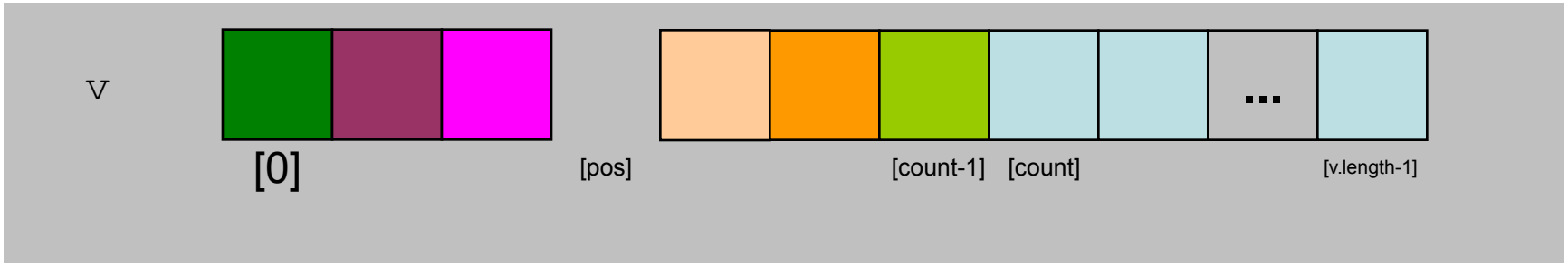


o elemento deverá ser removido da posição de índice `pos`.

Operações em Vectores

Remoção preservando a ordem dos que ficam

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

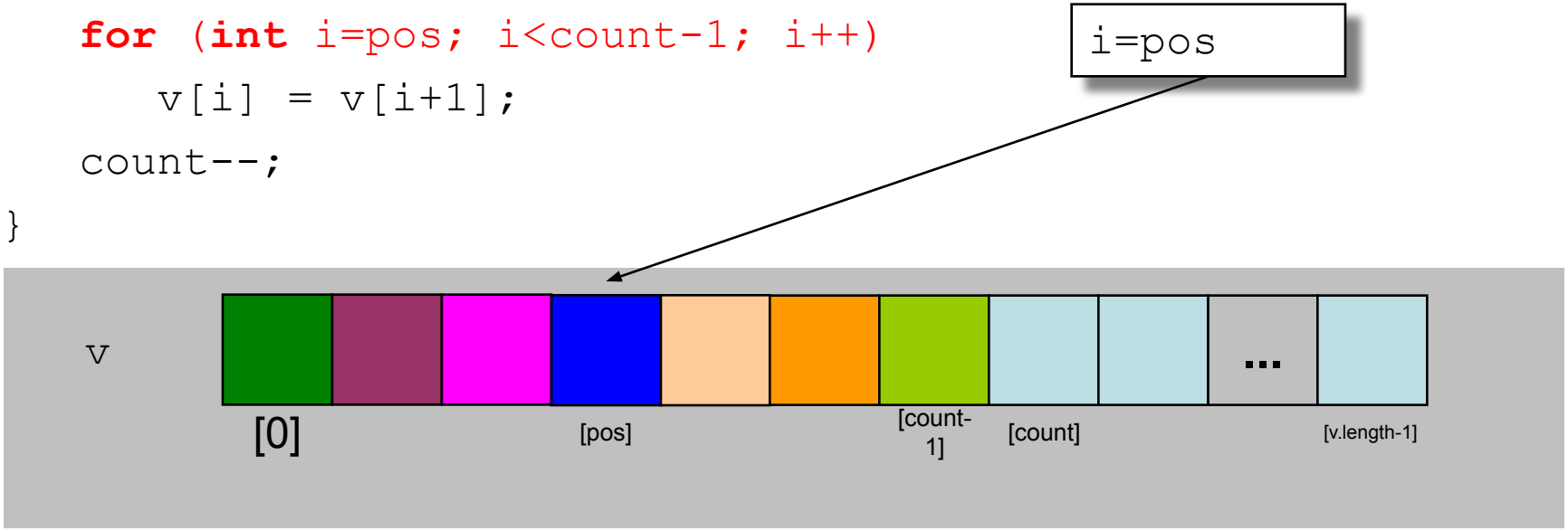


Mas não podemos deixar “um buraco” no vector...

Operações em Vectores

Remoção preservando a ordem dos que ficam

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

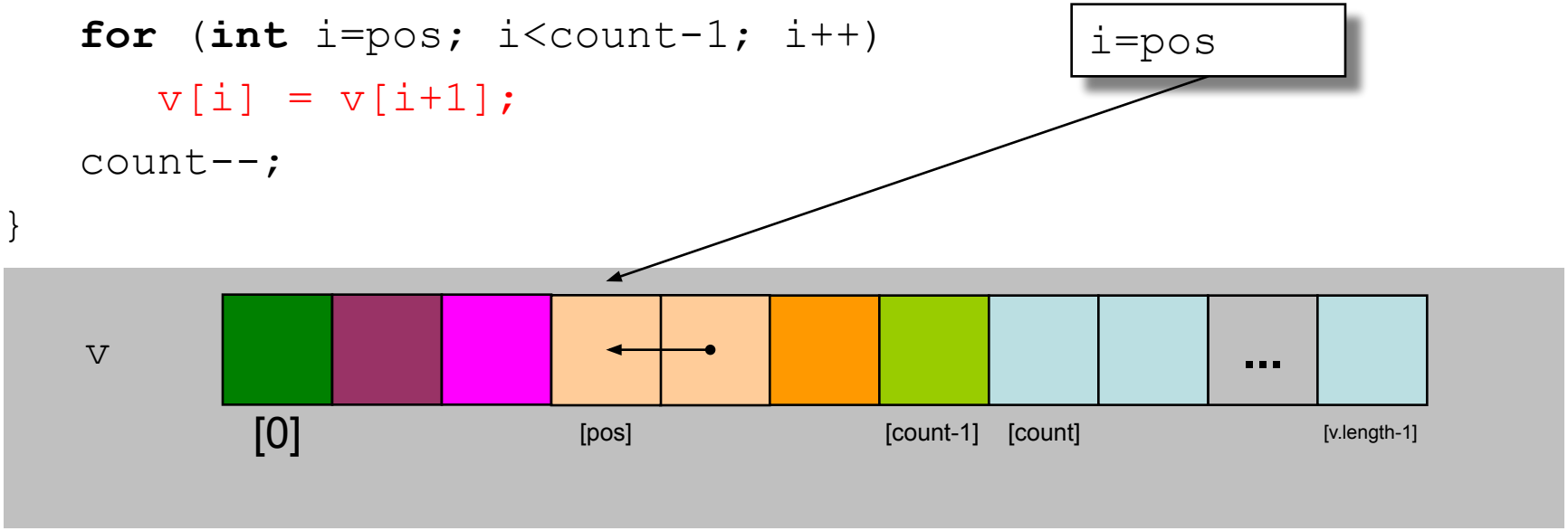


Mas não podemos deixar “um buraco” no vector...

Operações em Vectores

Remoção preservando a ordem dos que ficam

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

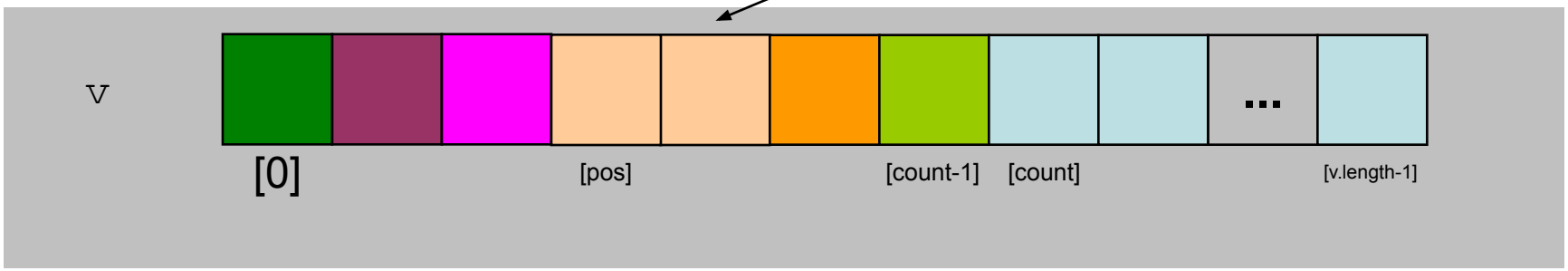


Mas não podemos deixar “um buraco” no vector...

Operações em Vectores

Remoção preservando a ordem dos que ficam

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

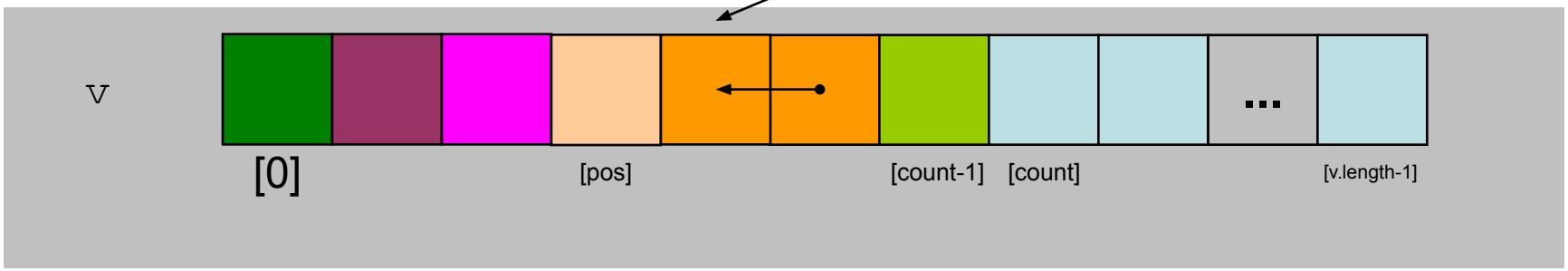


Mas não podemos deixar “um buraco” no vector...

Operações em Vectores

Remoção preservando a ordem dos que ficam

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

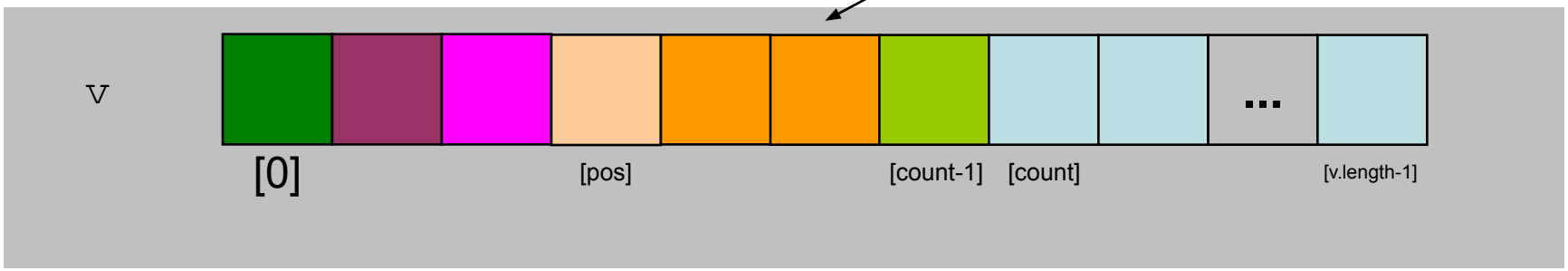


Mas não podemos deixar “um buraco” no vector...

Operações em Vectores

Remoção preservando a ordem dos que ficam

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

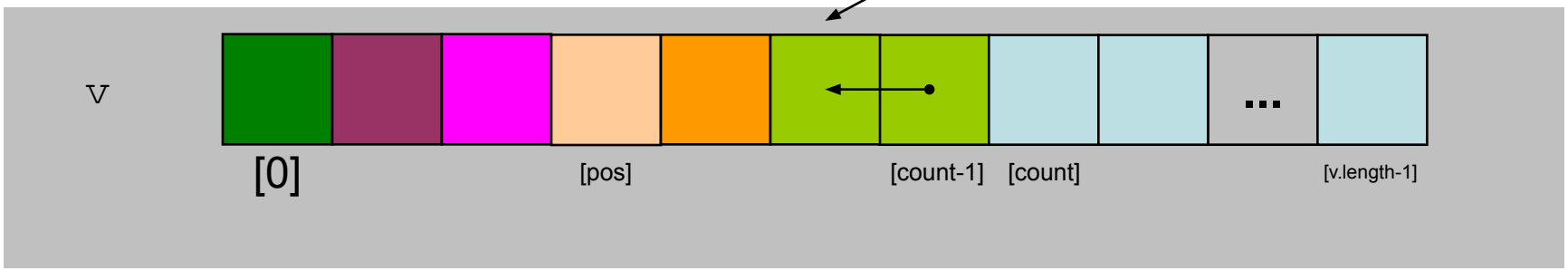


Mas não podemos deixar “um buraco” no vector...

Operações em Vectores

Remoção preservando a ordem dos que ficam

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

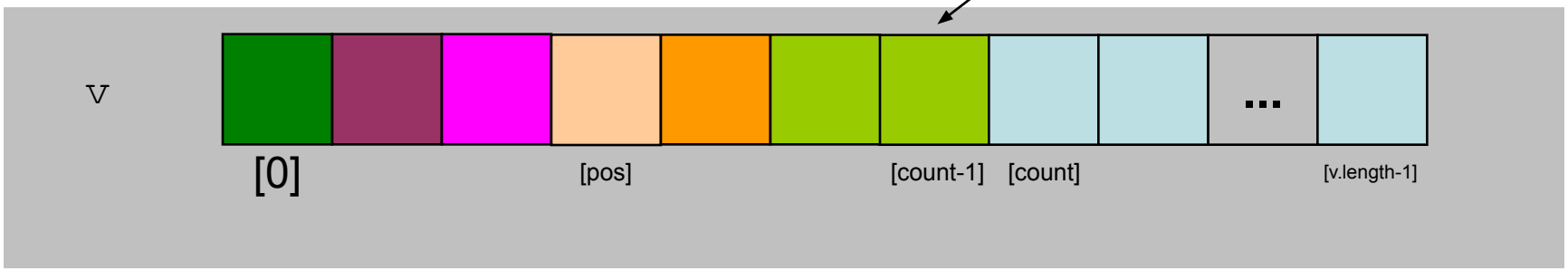


Mas não podemos deixar “um buraco” no vector...

Operações em Vectores

Remoção preservando a ordem dos que ficam

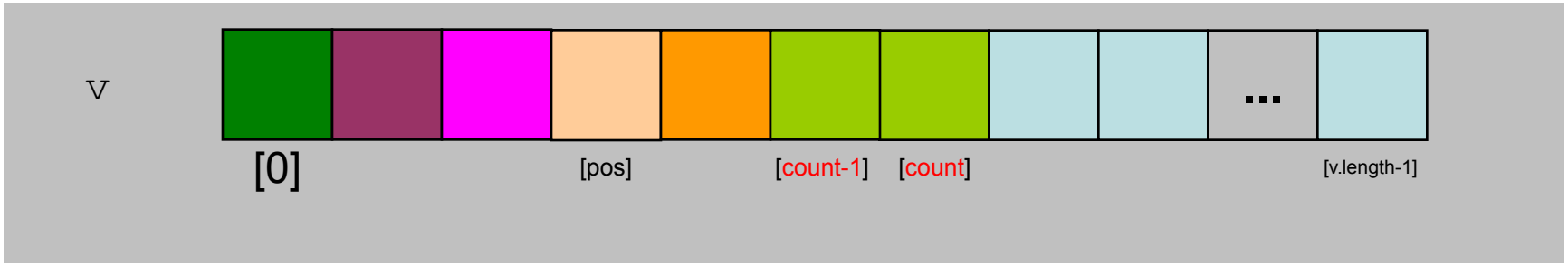
```
/** pre: 0 <= pos && pos < size() */
public void removeAt(int pos) {
    for (int i=pos; i<count-1; i++)
        v[i] = v[i+1];
    count--;
}
```



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

Operações em Vectores
Remoção preservando a ordem dos que ficam

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

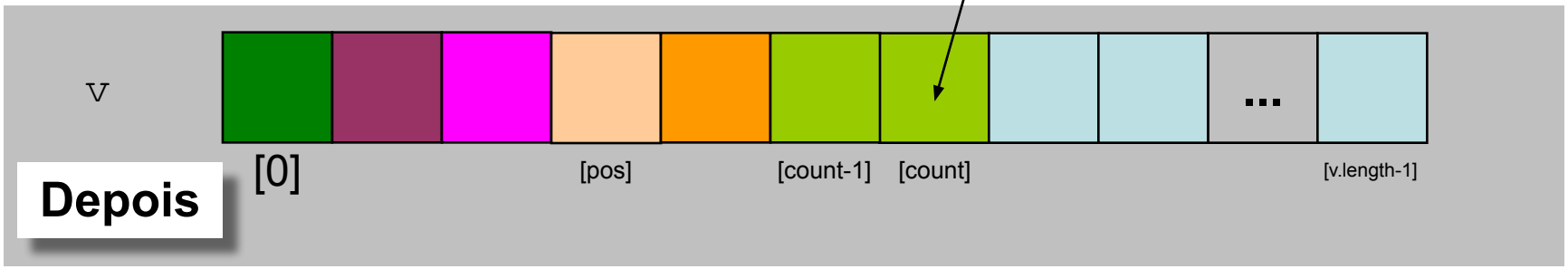


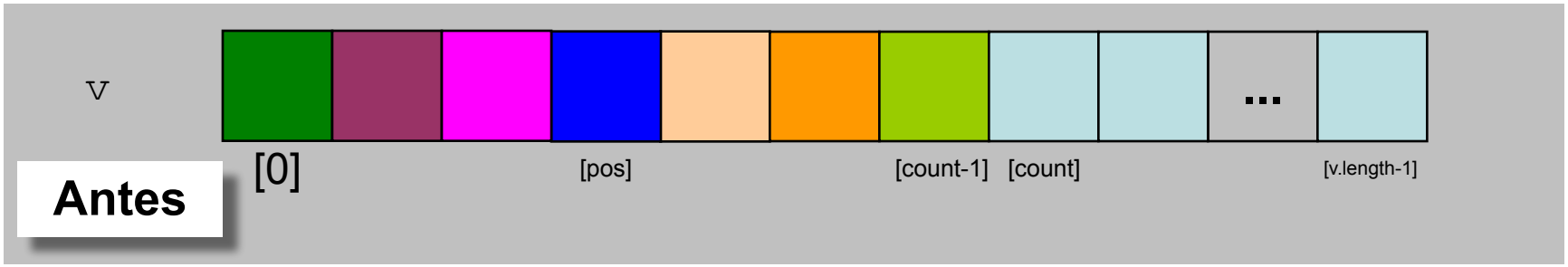
Operações em Vectores

Remoção preservando a ordem dos que ficam

```
/** pre: 0 <= pos && pos < size() */
public 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...





Operações em Vectores

Pesquisa Linear

Avaliação de Conjunções em Java

Avaliação de expr1 && expr2

A expressão expr1 é sempre avaliada.

- Se o valor de expr1 for true, a expressão expr2 é avaliada (porque ainda não se conhece o resultado da conjunção):

  true && true vale true

  true && false vale false

- Se o valor de expr1 for false, a expressão expr2 **não é avaliada** (porque já se conhece o resultado da conjunção, que é false):

  false && true vale false

  false && false vale false

Avaliação de Disjunções em Java

Avaliação de expr1 || expr2

A expressão expr1 é sempre avaliada.

- Se o valor de expr1 for false, a expressão expr2 é avaliada (porque ainda não se conhece o resultado da disjunção):

  false || true vale true

  false || false vale false

- Se o valor de expr1 for true, a expressão expr2 **não é avaliada** (porque já se conhece o resultado da disjunção, que é true):

  true || true vale true

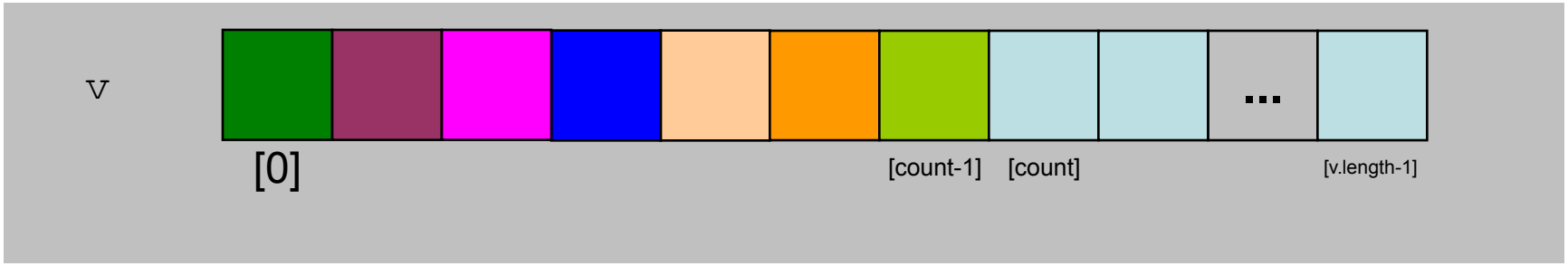
  true || false vale true

Pesquisa Linear – índices crescentes

```
public boolean searchForward(type e) {
    // Percorre-se o vector da esquerda para a direita.

    // Enquanto a posicao corrente for valida (nao se chegou ao fim) e
    // essa posicao nao tiver o elemento, passa-se 'a posicao seguinte.

    // Quando se sai do ciclo, distinguem-se os dois casos:
    // * Se a posicao for valida (não se chegou ao fim do vector),
    //   entao encontrou-se o elemento e devolve-se true.
    // * Se a posicao nao for valida (chegou-se ao fim do vector),
    //   entao o elemento não esta' no vector e devolve-se false.
}
```



Pesquisa Linear – índices crescentes

```
public boolean searchForward(type e) {
    // Percorre-se o vector da esquerda para a direita - inicializacao.
    int i = 0;

    // Enquanto a posicao corrente for valida (nao se chegou ao fim) e
    // essa posicao nao tiver o elemento, passa-se 'a posicao seguinte.

    // Quando se sai do ciclo, distinguem-se os dois casos:
    // * Se a posicao for valida (não se chegou ao fim do vector),
    //   entao encontrou-se o elemento e devolve-se true.
    // * Se a posicao nao for valida (chegou-se ao fim do vector),
    //   entao o elemento não esta' no vector e devolve-se false.
}
```

Pesquisa Linear – índices crescentes

```
public boolean searchForward(type e) {
    // Percorre-se o vector da esquerda para a direita - inicializacao.
    int i = 0;

    // Condicao: a posicao corrente e' valida (nao se chegou ao fim) e
    // essa posicao nao tem o elemento.
    while (i < count && v[i] != e)
        // Passa-se 'a posicao seguinte.

    // Quando se sai do ciclo, distinguem-se os dois casos:
    // * Se a posicao for valida (não se chegou ao fim do vector),
    //   entao encontrou-se o elemento e devolve-se true.
    // * Se a posicao nao for valida (chegou-se ao fim do vector),
    //   entao o elemento não esta' no vector e devolve-se false.
}
```

Só se pode aceder à posição **`i`** do vector (**`v[i]`**) depois de garantir que **`0 ≤ i < count`**

Pesquisa Linear – índices crescentes

```
public boolean searchForward(type e) {
    // Percorre-se o vector da esquerda para a direita - inicializacao
    int i = 0;

    // Condicao: a posicao corrente e' valida (nao se chegou ao fim) e
    // essa posicao nao tem o elemento.
    while (i < count && v[i] != e)
        // Passa-se 'a posicao seguinte
        i++;

    // Quando se sai do ciclo, distinguem-se os dois casos:
    // * Se a posicao for valida (não se chegou ao fim do vector),
    //   entao encontrou-se o elemento e devolve-se true.
    // * Se a posicao nao for valida (chegou-se ao fim do vector),
    //   entao o elemento não esta' no vector e devolve-se false;
}
```

Só se pode aceder à posição **`i`** do vector (**`v[i]`**) depois de garantir que **`0 ≤ i < count`**

Pesquisa Linear – índices crescentes

```
public boolean searchForward(type e) {
    // Percorre-se o vector da esquerda para a direita - inicializacao
    int i = 0;
    // Enquanto a posicao corrente for valida (nao se chegou ao fim) e
    // essa posicao nao tiver o elemento, passa-se 'a posicao seguinte
    while (i < count && v[i] != e)
        i++;

    // Quando se sai do ciclo, distinguem-se os dois casos:
    if (i < count) // v[i] == e
        // Se a posicao e´ valida, encontrou-se o elemento na posicao i.
        return true;
    else // i == count
        // Percorreu-se o vector sem se ter encontrado o elemento.
        return false;
}
```

Pesquisa Linear – índices crescentes

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

Primeira Versão

Será possível simplificar a parte final?

Pesquisa Linear – índices crescentes

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

Primeira Versão
Será possível simplificar a parte final?

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

Versão Final

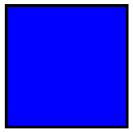
Pesquisa Linear – índices crescentes

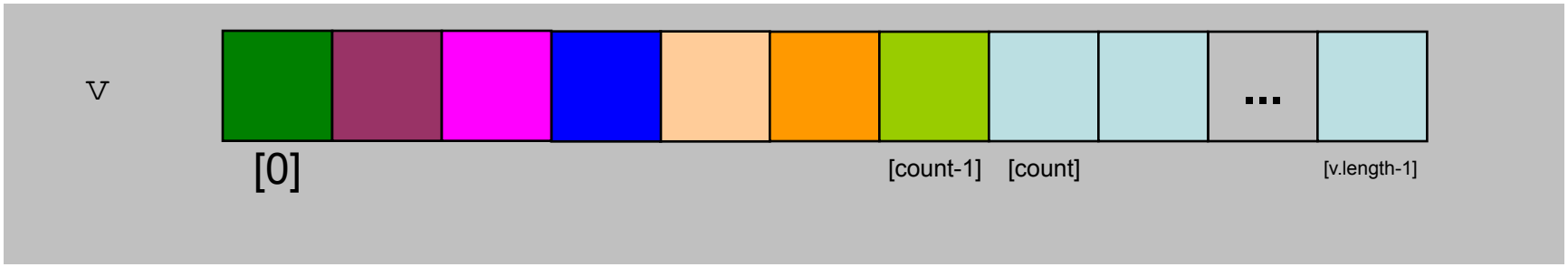
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
     i++;

   return i < count;
 }
```

Existirá no vector o elemento

e =  ?



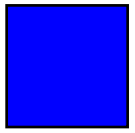
Pesquisa Linear – índices crescentes

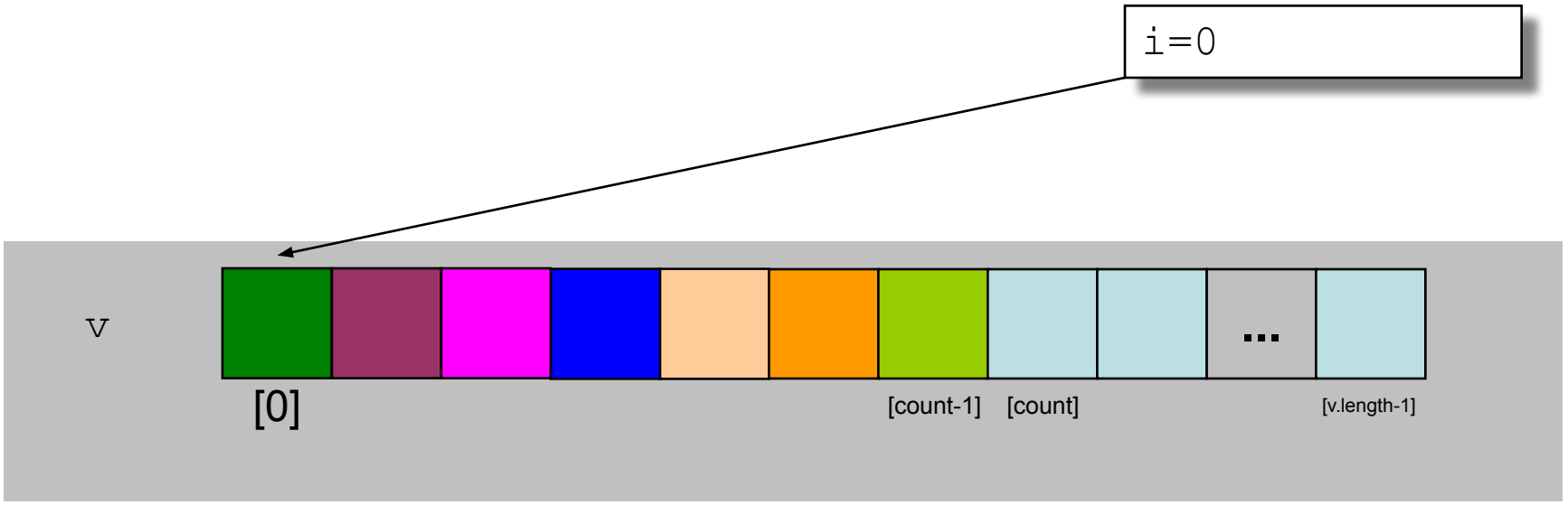
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e =  ?



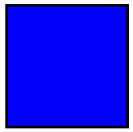
Pesquisa Linear – índices crescentes

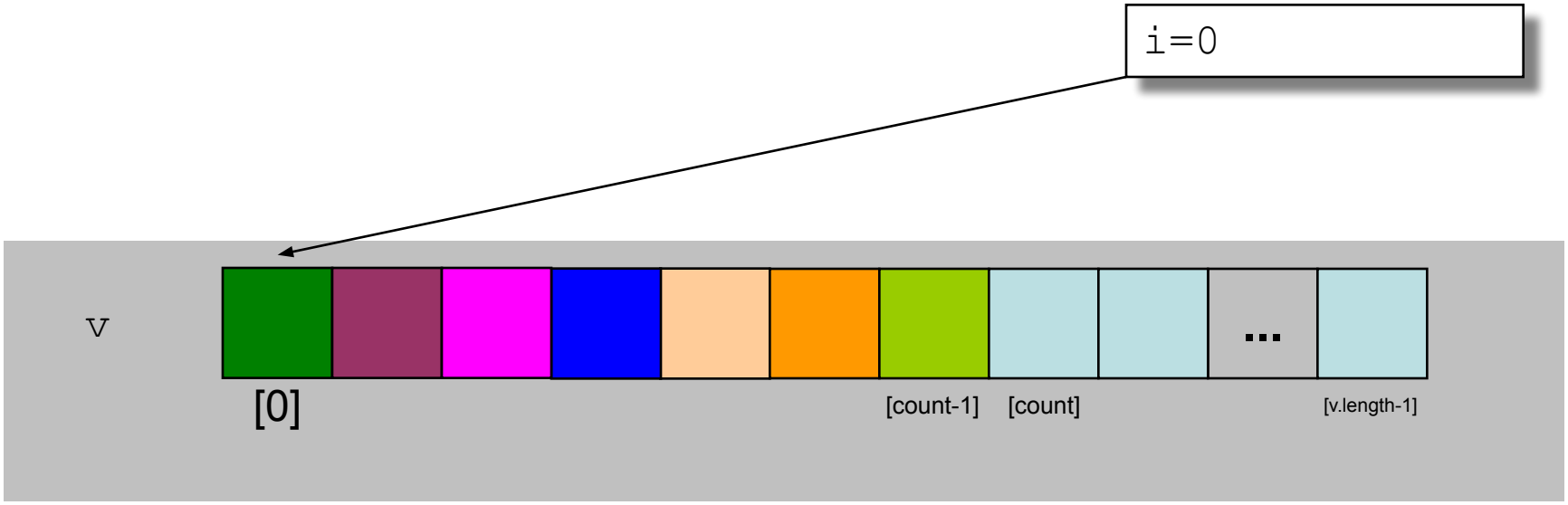
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e =  ?



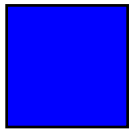
Pesquisa Linear – índices crescentes

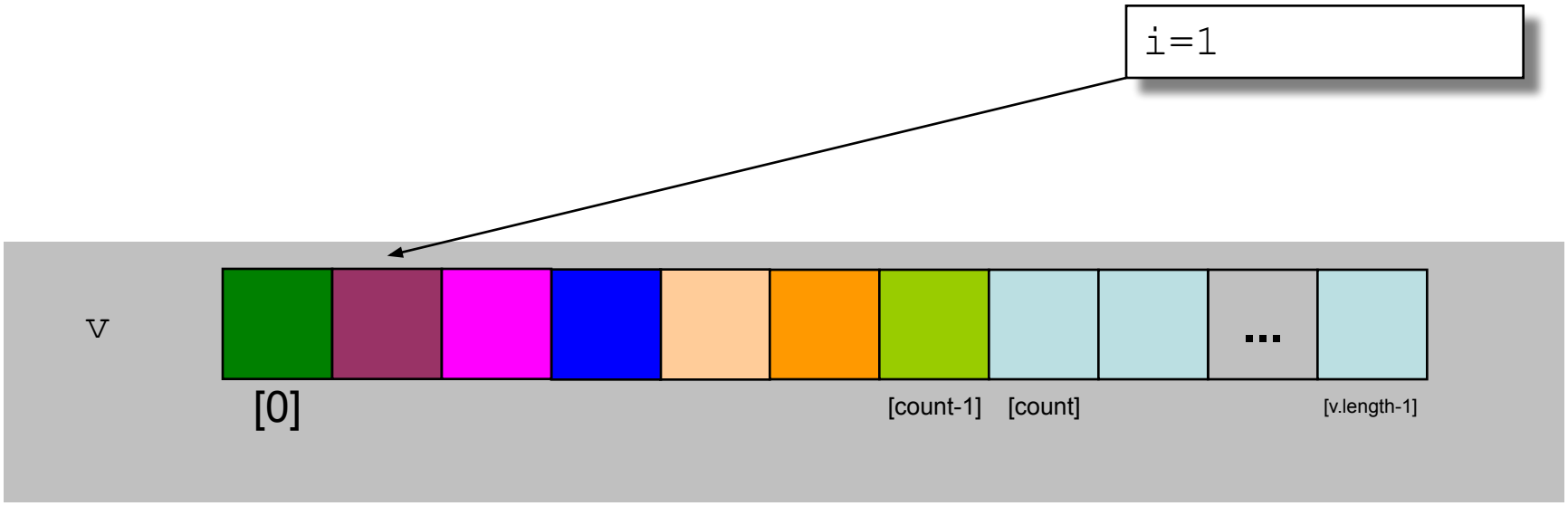
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e =  ?



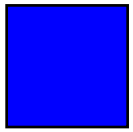
Pesquisa Linear – índices crescentes

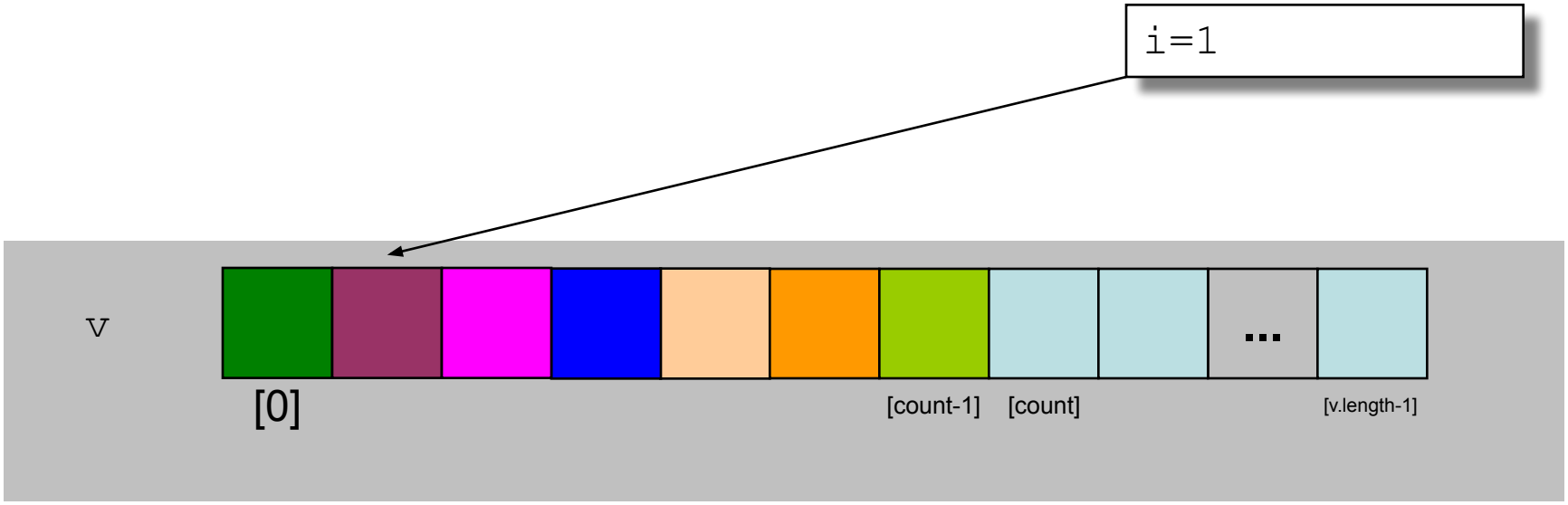
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e =  ?



Pesquisa Linear – índices crescentes

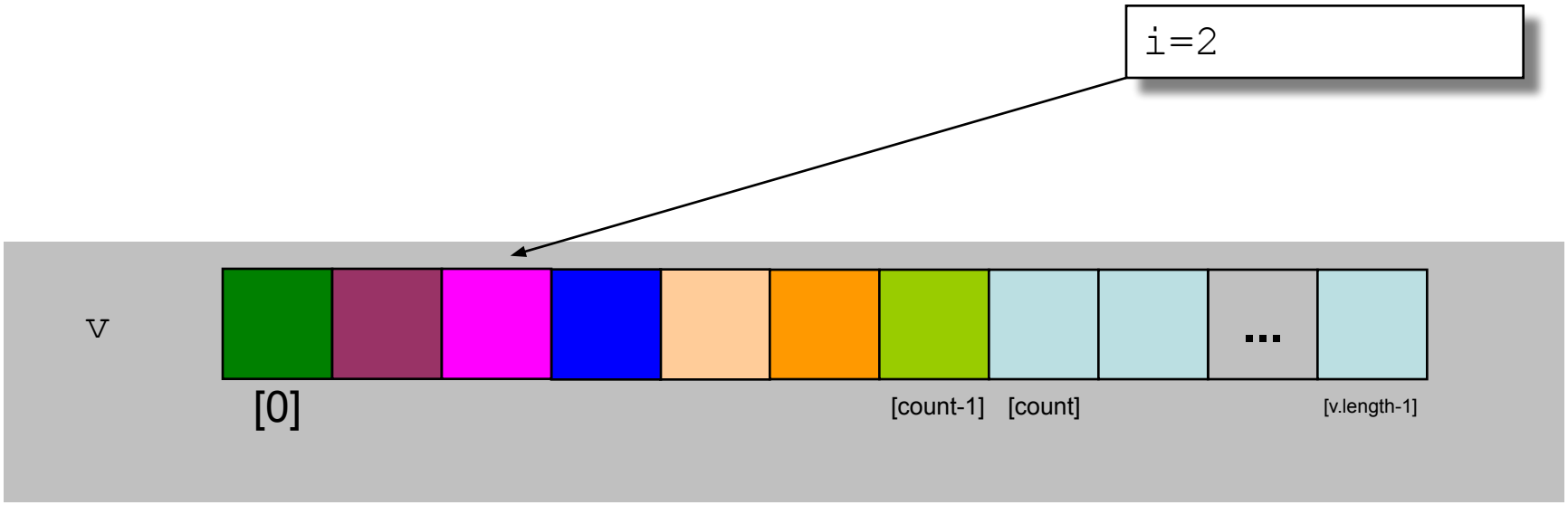
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e = ?



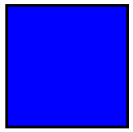
Pesquisa Linear – índices crescentes

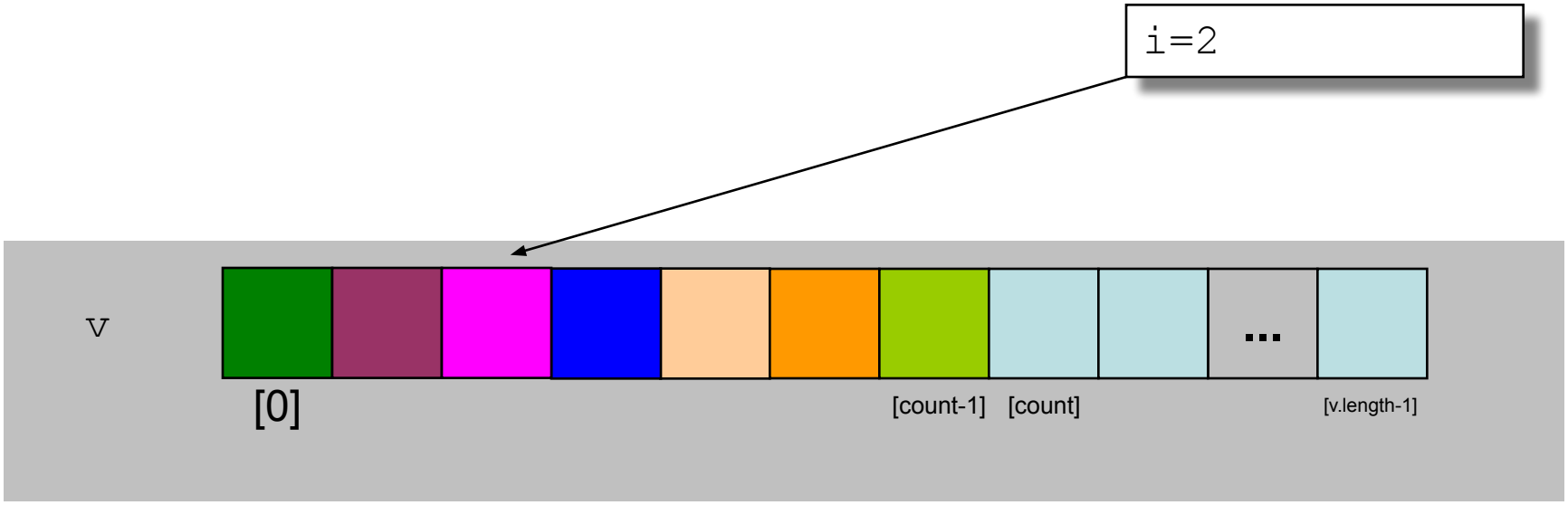
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e =  ?



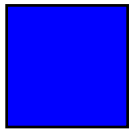
Pesquisa Linear – índices crescentes

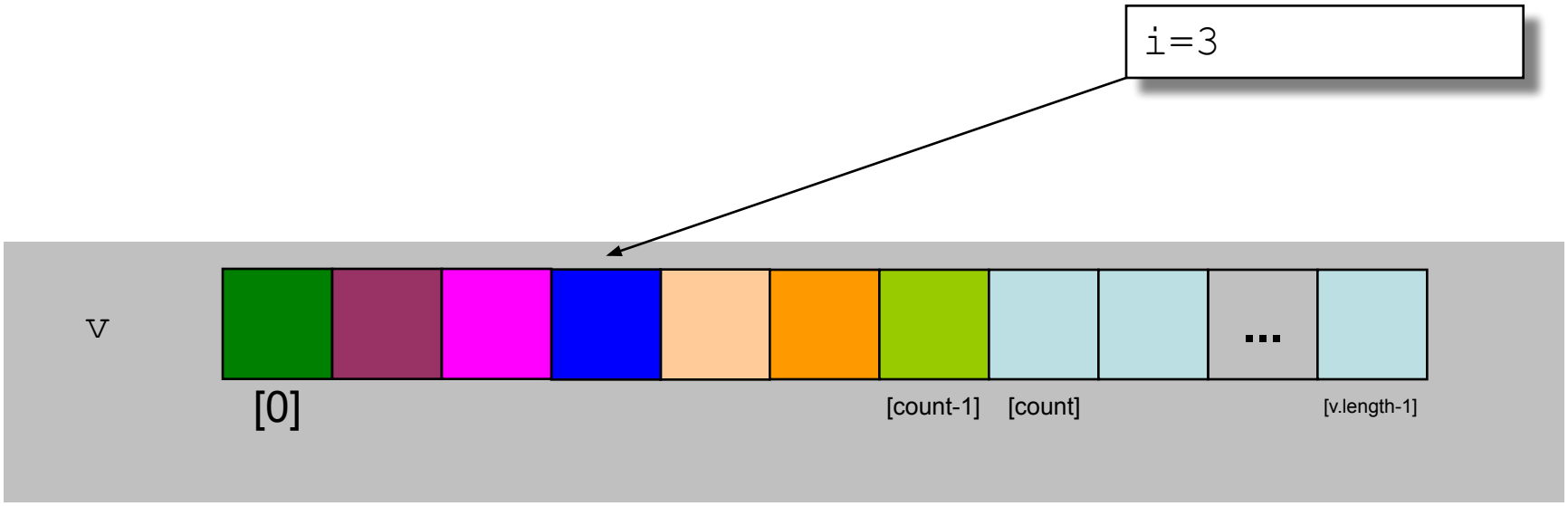
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e =  ?



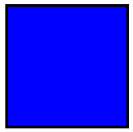
Pesquisa Linear – índices crescentes

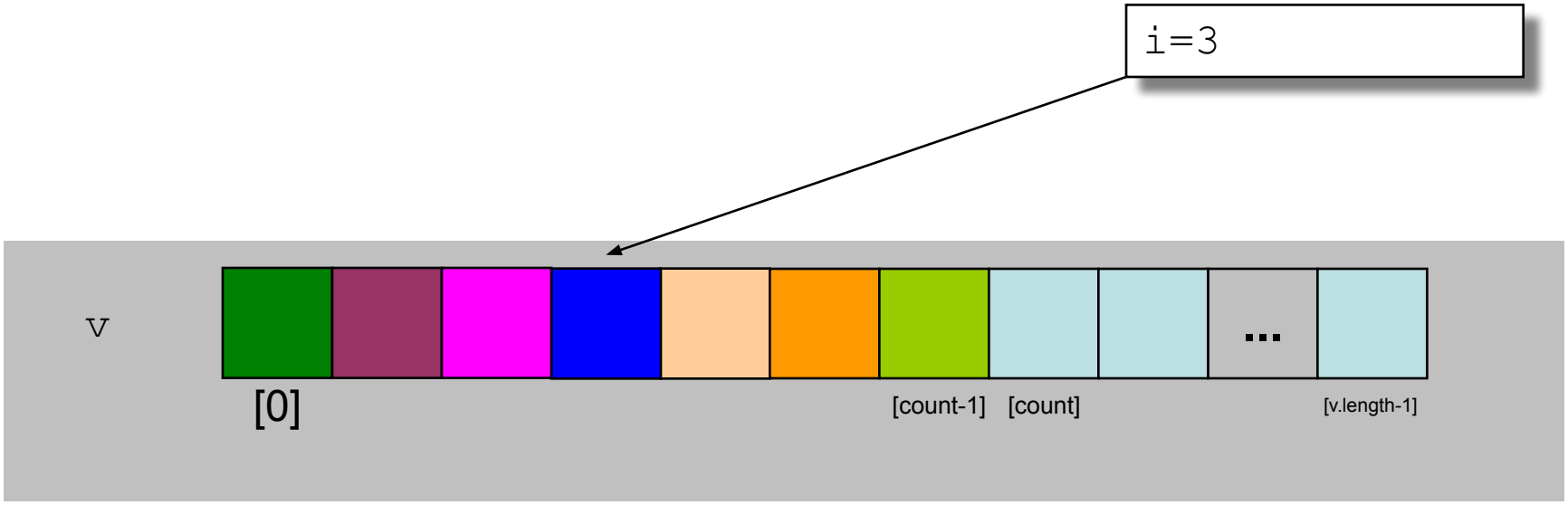
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e =  ?



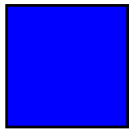
Pesquisa Linear – índices crescentes

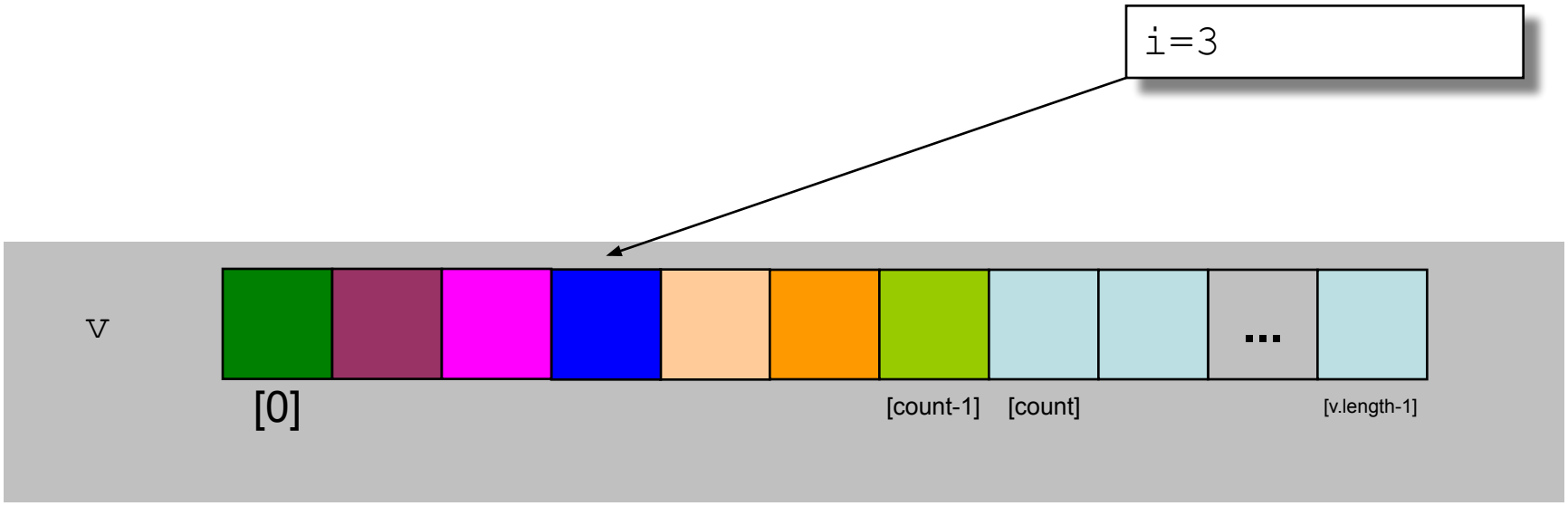
```
public boolean searchForward(type e) {
   int i=0;

   while (i < count && v[i] != e)
      i++;

   return i < count;
}
```

Existirá no vector o elemento

e =  ?



Pesquisa Linear – índices crescentes

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

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.

Pesquisa Linear – índices crescentes

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

Muitas vezes, não basta saber se o elemento ocorre no vector; pretende-se determinar a sua localização.

Se o elemento ocorrer no vector (e o vector for percorrido da esquerda para a direita), retorna-se a menor posição onde o elemento ocorre.

Se o elemento não ocorrer no vector, retorna-se **-1**
(que nunca é uma posição válida).

Pesquisa Linear – índices crescentes

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

Mas o melhor será programar duas funções:

- a função mais geral (`searchIndexOf`) que determina a posição do elemento dentro do vector;
- a função booleana de busca (`searchForward`) à custa da primeira.

```
public boolean searchForward(...) {
    return this.searchIndexOf(...) >= 0;
}
```

Pesquisa Linear – índices decrescentes

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

Primeira Versão
Será possível simplificar a parte final?

Se o elemento ocorrer no vector (e o vector for percorrido **da direita para a esquerda**), retorna-se a **maior** posição onde o elemento ocorre.

Se o elemento não ocorrer no vector, retorna-se **-1** (que nunca é uma posição válida).

Pesquisa Linear – índices decrescentes

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

Primeira Versão
Será possível simplificar a parte final?

```
public int searchLastIndexOf(...) {
   int i=count-1;
   while (i >= 0 && !p(v[i]))
     i--;
   return i;
}
```

Versão Final

Inserção em Vector Ordenado

```
public void insertSort(type e) {
    int i=0;
    while (i < count && v[i] < e)
        i++;
    this.insertAt(e, i);
}
```

Cada vez que se insere, procura-se a posição correcta para o elemento de forma a preservar a ordem dos elementos no vector.

Procura-se o primeiro elemento com um valor superior ou igual a `e`. A inserção é feita na posição desse elemento.