Estruturas de Informação I

3. Listas baseadas em Listas Encadeadas

João Araujo Ribeiro

jaraujo@uerj.br

Departamento de Engenharia de Sistemas e Computação

Universidade do Estado do Rio de Janeiro

Resumo

Nesta aula vamos estudar a implementação da interface de lista usando listas encadeadas.

http://www.opendatastructures.org/ods-python/3_Linked_Lists.html

3 - Listas Encadeadas

- 3.1 Listas Simplesmente Encadeadas- SLList
- 3.2 Listas Duplamente Encadeadas
- 3.3 Lista Encadeada Eficiente para espaço

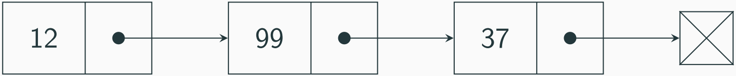
Listas Encadeadas

Nas listas sequenciais anteriores é necessário um grande esforço computacional para operações de inserção e remoção de nós.

Uma forma de permitir o crescimento dinâmico do comprimento máximo de uma lista é representar a lista por *encadeamento*, onde os nós são ligados entre si para indicar a relação de ordem entre eles.

Encadeamento

Cada nó deve conter não apenas o dado mas também a indicação do nó seguinte, casa haja algum. Neste caso, o encadeamento é lógico.



A Lista Linear agrupa informações referentes a um conjunto de elementos que, de alguma forma, se relacionam entre si.

Listas Encadeadas

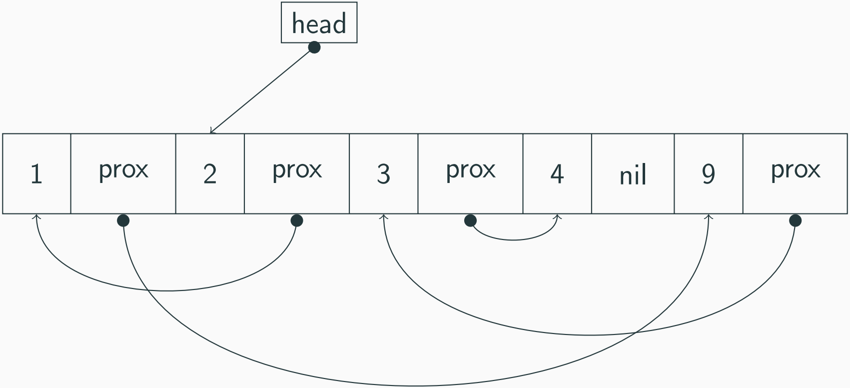
- Em listas encadeadas, elementos consecutivos na lista não implicam em elementos consecutivos na representação (a ordem é lógica).

Listas Encadeadas

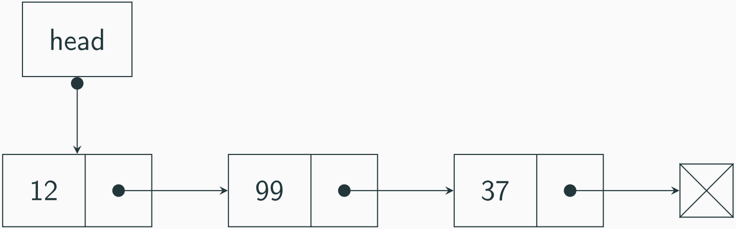
- Em listas encadeadas, elementos consecutivos na lista não implicam em elementos consecutivos na representação (a ordem é lógica).
- Na implementação é necessário armazenar separadamente a informação de um elemento da lista, normalmente o primeiro.

Listas Encadeadas

- Em listas encadeadas, elementos consecutivos na lista não implicam em elementos consecutivos na representação (a ordem é lógica).
- Na implementação é necessário armazenar separadamente a informação de um elemento da lista, normalmente o primeiro.
- Existem duas formas de se representar listas encadeadas, através de array, denominadas listas estáticas, ou por ponteiros chamadas listas dinâmicas.


head
1
prox
2
prox
3
prox
4
nil
9
prox

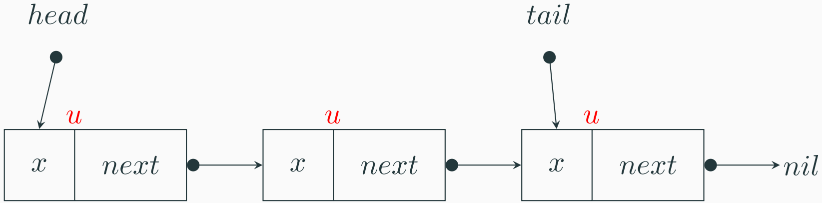
Lista Dinâmica



3 - Listas Encadeadas

3.1 Listas Simplesmente Encadeadas-SLList

Uma $SLList$ (lista simplesmente encadeada) é uma sequência de *Nós*. Cada nó u armazena um valor de dados $u.x$ e uma referência $u.next$ para o próximo nó na sequência. Para o último nó w na sequência, $w.next = null$



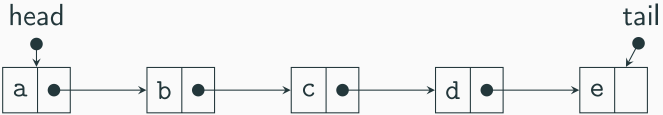
SLList - initialize

Para eficiência, um $SLList$ usa as variáveis $head$ e $tail$ para manter o registro do primeiro e do último nó na sequência, bem como um número inteiro n para acompanhar o tamanho da sequência:

initialize()
    $n \leftarrow 0$
    $head \leftarrow nil$
    $tail \leftarrow nil$

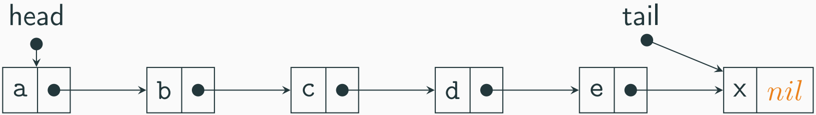
Sequência de operações de fila e pilha usando LSE

$add(\mathtt{x})$



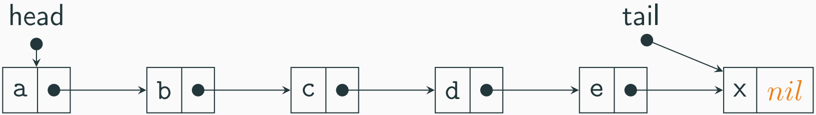
Sequência de operações de fila e pilha usando LSE

$add(\texttt{x})$



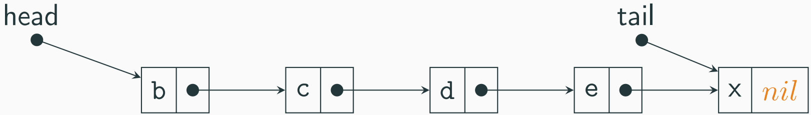
Sequência de operações de fila e pilha usando LSE

$remove()$



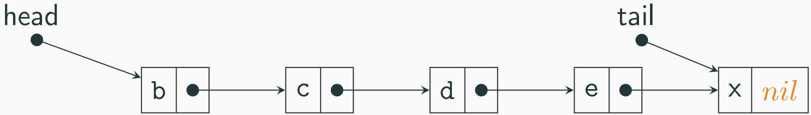
Sequência de operações de fila e pilha usando LSE

$remove()$



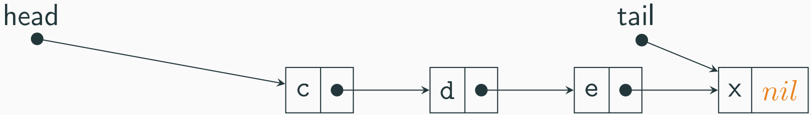
Sequência de operações de fila e pilha usando LSE

$pop()$



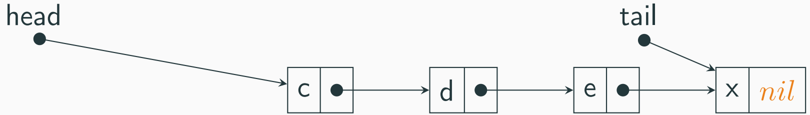
Sequência de operações de fila e pilha usando LSE

$pop()$



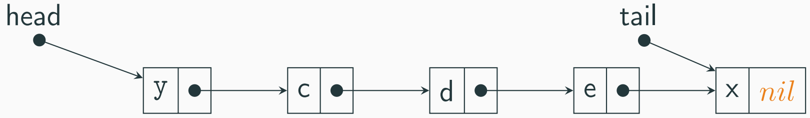
Sequência de operações de fila e pilha usando LSE

$push(\mathrm{y})$

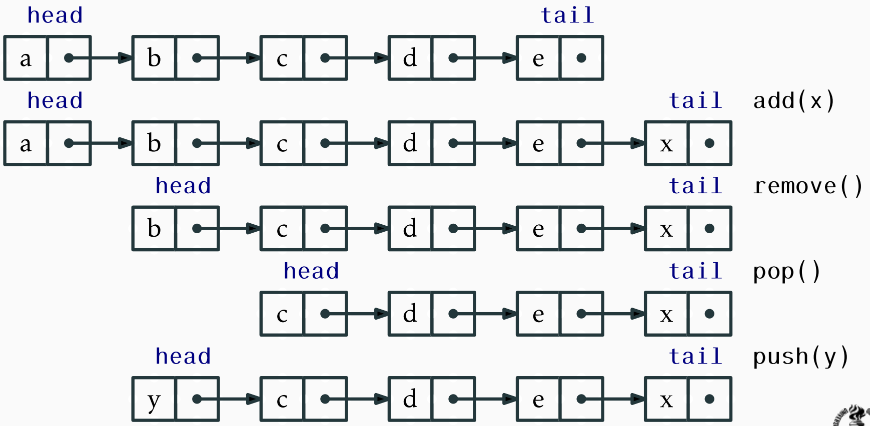


Sequência de operações de fila e pilha usando LSE

$push(\mathrm{y})$



Sequência de operações de fila e pilha usando LSE



Pilha com SLL

Uma $SLList$ pode implementar eficientemente as operações $push()$ e $pop()$ de uma $Stack$, adicionando e removendo elementos na cabeça da sequência.

$push()$

A operação $push()$ simplesmente cria um novo nó u com valor de dados x, define $u.next$ no cabeçalho antigo da lista e torna u o novo cabeçalho da lista. Finalmente, ele incrementa n, uma vez que o tamanho da $SLList$ aumentou em um:

$\mathrm{push}(x)$
  $u \leftarrow \mathrm{new\_node}(x)$
  $u.next \leftarrow head$
  $head \leftarrow u$
  if $n = 0$ **then**
    $tail \leftarrow u$
  $n \leftarrow n + 1$
  return x

A operação $pop()$, depois de verificar que a $SLList$ não está vazia, remove a cabeça definindo $head = head.next$ e decrementando n. Um caso especial ocorre quando o último elemento está sendo removido, caso em que $tail$ é definido como nil:

pop()
    if $n = 0$ **then return** nil
    $x \leftarrow head.x$
    $head \leftarrow head.next$
    $n \leftarrow n - 1$
    if $n = 0$ **then**
        $tail \leftarrow nil$
    return x

Fila com SLList

Uma $SLList$ também pode implementar as operações de fila FIFO, $add(x)$ e $remove()$, em tempo constante.

Remoções são feitas a partir da cabeça da lista e são idênticas à operação de $\mathrm{pop}()$:

```
remove()
    return pop()
```

$add()$

Adições, por outro lado, são feitas no final da lista. Na maioria dos casos, isso é feito definindo $tail.next = u$, onde u é o nó recém-criado que contém x. No entanto, um caso especial ocorre quando $n = 0$, caso em que $tail = head = null$.

$\mathrm{add}(x)$
    $u \leftarrow \mathrm{new\_node}(x)$
    if $n = 0$ **then**
        $head \leftarrow u$
    else
        $tail.next \leftarrow u$
    $tail \leftarrow u$
    $n \leftarrow n + 1$
    return $true$

Uma $SLList$ implementa a interface para $Stack$ e (FIFO) $Queue$. As operações $push(x)$, $pop()$, $add(x)$ e $remove()$ são executadas em um tempo $O(1)$ por operação.

Uma $SLList$ quase implementa o conjunto completo de operações de uma $Deque$.

A única operação que falta é a remoção da cauda de uma $SLList$.

Remover a cauda de uma $SLList$ é difícil porque requer a atualização do valor da $tail$ para que ele aponte para o nó w que precede $tail$ na $SLList$; este é o nó w tal que $w.next = tail$.

Infelizmente, a única maneira de chegar ao w é atravessar a $SLList$ começando em $head$ e tomando $n - 2$ passos.

3 - Listas Encadeadas

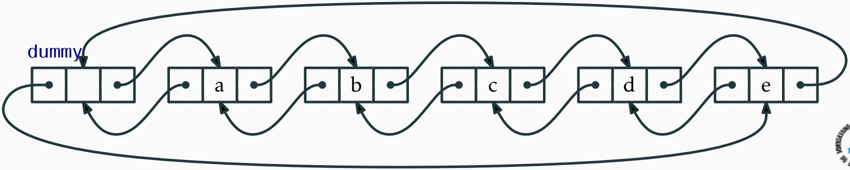
3.2 Listas Duplamente Encadeadas

DLList - Listas Duplamente Encadeadas

A $DLList$ (lista duplamente encadeada) é muito semelhante a uma $DLList$, exceto que cada nó u em uma $DLList$ tem referências tanto ao nó $u.next$ que o sucede, quanto ao nó $u.prev$ que o precede.

Quando implementar uma $SLList$, percebemos que sempre temos vários casos especiais para se preocupar. Por exemplo, remover o último elemento ou adicionar um elemento vazio a uma $SLList$ requer cuidado para garantir que $head$ e $tail$ sejam atualizados corretamente. Numa $DLList$, o número destes casos especiais aumenta consideravelmente.

Talvez a maneira mais simples de lidar com todos os casos especiais em uma LDE seja criar um nó $dummy$. Este nó não contém nenhum dado, porém age como um sentinela de modo que não existam nós especiais; cada nó possui ambos $next$ e $prev$, com $dummy$ atuando como um nó que é o próximo após o último nó e é o que precede o primeiro nó na lista. Deste modo, os nós da lista são ligados em um círculo.



initialize()
$n \leftarrow 0$
$dummy \leftarrow$ DLList.Node(nil)
$dummy.prev \leftarrow dummy$
$dummy.next \leftarrow dummy$

Encontrar um nó com um índice em uma LDE é fácil; podemos ou iniciar a busca pela cabeça da lista ($dummy.next$) e avançar, ou iniciar a busca pela cauda da lista ($dummy.prev$) e recuar. Isto nos permite chegar ao $i$ésimo nó em um tempo de $O(1 + \min\{i, n - i\})$:

Obter i

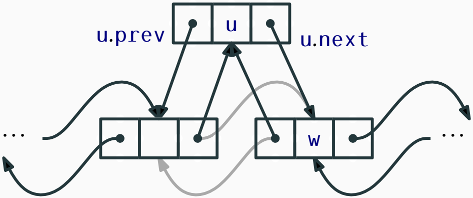
```
get_node(i)
    if i < n/2 then
        p ← dummy.next
        repeat i times
            p ← p.next
    else
        p ← dummy
        repeat n − i times
            p ← p.prev
    return p
```

$get(i)$ e $set(i, x)$

get(i)
    return get_node(i).x

set(i, x)
    $u \leftarrow$ get_node(i)
    $y \leftarrow u.x$
    $u.x \leftarrow x$
    return y

Se temos a referência para um nó w em uma LDE e queremos inserir um nó u antes de w, então isto é apenas uma questão de fazer $u.next = w$, $u.prev = w.prev$, e depois ajustando $u.prev.next$ e $u.next.prev$. Graças ao nó dummy, não precisamos nos preocupar se $w.prev$ ou $w.next$ existam ou não.



```
add_before(w, x)
    u ← ListaDE.Node(x)
    u.prev ← w.prev
    u.next ← w
    u.next.prev ← u
    u.prev.next ← u
    n ← n + 1
    return u
```

$add(i, x)$

$\text{add}(i, x)$
    $\text{add\_before}(\text{get\_node}(i), x)$

Remover um nó w de uma LDE é fácil. Precisamos somente ajustar os ponteiros em $w.next$ e $w.prev$ de modo que eles pulem w. Novamente, o uso do nó dummy elimina a necessidade de considerar qualquer caso especial:

$\text{remove}(w)$
  $w.prev.next \leftarrow w.next$
  $w.next.prev \leftarrow w.prev$
  $n \leftarrow n - 1$

Agora a operação $\text{remove}(i)$ é trivial. Encontramos o nó com índice i e o removemos:

$$\begin{array}{l} \text{remove}(i) \\ \quad \text{remove}(\text{get\_node}(i)) \end{array}$$

3 - Listas Encadeadas

3.3 Lista Encadeada Eficiente para espaço

Lista Encadeada Eficiente para espaço

Uma das desvantagens das listas encadeadas (além do tempo que leva para acessar elementos) é o seu uso de espaço. Cada nó em uma LDE requer duas referências adicionais para os nós $next$ e $previous$ na lista. Dois dos campos em um nó são dedicados a manter a lista, enquanto somente um campo contém dados!

Uma LEE reduz o espaço desperdiçado usando uma ideia simples: Em vez de armazenar elementos individuais em uma LDE, armazenamos um bloco (array) contendo vários itens. Sendo mais preciso, uma LEE é parametrizada por um tamanho de bloco b. Cada nó individual node em uma LEE armazena um bloco que pode conter até $b + 1$ elementos.

BDeque como infraestrutura

Por razões que ficarão claras mais tarde, seria útil se pudéssemos fazer operações de Deque em cada bloco. A estrutura de dados que escolhemos é uma BDeque (bounded deque), derivada de uma estrutura ArrayDeque. Uma BDeque difere de uma ArrayDeque em um pequeno detalhe: Quando uma BDeque é criada, o tamanho da array de suporte a é fixado em $b+1$ e nunca cresce ou diminui. A propriedade importante de uma BDeque é que ela permite a adição ou remoção de elementos seja na frente seja nos fundos em um tempo constante. Isto será útil quando os elementos são deslocados de um bloco para outro.

BDeque como infraestrutura

Uma LEE é somente uma LDE de blocos. Além dos ponteiros $next$ e $prev$, cada nó u em uma LEE contém uma BDeque, $u.d$.

Restrições de Espaço

Uma LEE possui restrições severas sobre o número de elementos dentro de um bloco: A menos que um bloco seja o último, então esse bloco contém pelo menos $b-1$ e no máximo $b+1$ elementos. Isto significa que, se uma LEE contém n elementos, então ele contém ao menos

$$n/(b-1)+1 = O(n/b)$$

blocos.

Encontrando um elemento

O primeiro desafio com uma LEE é encontrar o elemento da lista com índice i. Note que a localização do elemento consiste em duas partes:

O nó u que contém o bloco que contém o elemento com índice i; e o índice j do elemento dentro do seu bloco.

Para encontrar o bloco que contém um elemento em particular, fazemos do mesmo modo que fizemos para uma LDE. Começamos ou da frente da lista e atravessamos avançando, ou começamos por trás e atravessamos recuando na lista até encontrarmos o nó que queremos. A única diferença é que, a cada vez que nos movemos de um bloco para outro, pulamos um bloco inteiro de elementos.

$get\_location()$

```
get_location(i)
    if i < n ÷ 2 then
        u ← dummy.next
        while i ≥ u.d.size() do
            i ← i − u.d.size()
            u ← u.next
        return u, i
    else
        u ← dummy
        idx ← n
        while i < idx do
            u ← u.prev
            idx ← idx − u.d.size()
    return u, i − idx
```

$get()$ e $set()$

get(i)

  $u, j \leftarrow$ get_location(i)

  return $u.d.$get(j)

set(i, x)

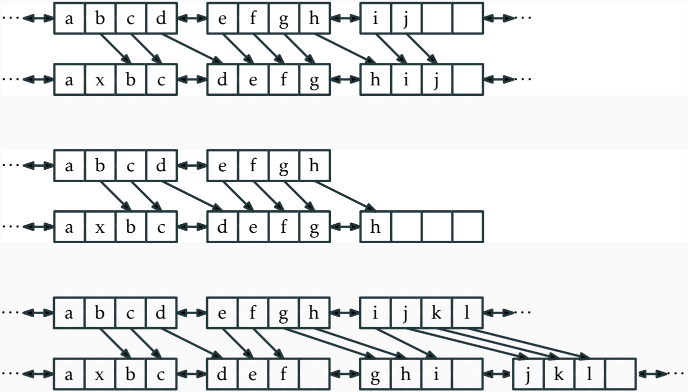
  $u, j \leftarrow$ get_location(i)

  return $u.d.$set(j, x)

$append(x)$ - Adicionando elemento no final da lista

append(x)
    $last \leftarrow dummy.prev$
    if $last = dummy$ **or** $last.d.\text{size}() = b + 1$ **then**
        $last \leftarrow \text{add\_before}(dummy)$
    $last.d.\text{append}(x)$
    $n \leftarrow n + 1$

$add(i,x)$ - Adicionando elemento no meio da lista



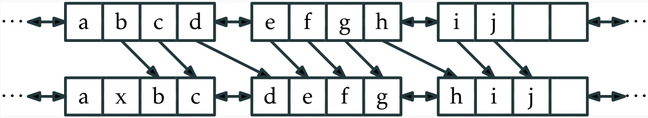
a

b

c

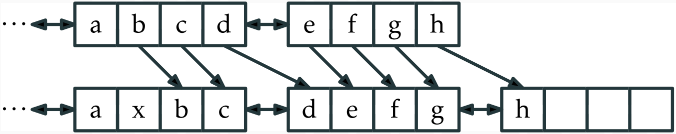
$add(i, x)$ - Caso a

Rapidamente (em $r + 1 \leq b$ passos) encontramos um nó u_r cujo bloco não está cheio. Neste caso, executamos r deslocamentos de um elemento de um bloco para o próximo, de modo que o espaço livre em u_r se torna um espaço livre em u_0. Podemos agora inserir x no bloco u_0.



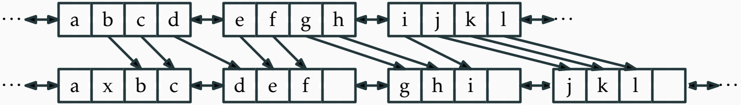
$add(i, x)$ - Caso b

Rapidamente (em $r + 1 \leq b$ passos) encontramos o fim da lista de blocos. neste caso, adicionamos um novo bloco no fim da lista e prosseguimos como no primeiro caso.



$add(i,x)$ - Caso c

Após b passos não encontramos qualquer bloco que não esteja cheio. Neste caso, $u_0, \ldots, u_{b-1}$ é uma sequência de b blocos em que cada um contém $b+1$ elementos. Inserimos um novo bloco u_b ao final desta sequência e *espalhamos* os elementos originais $b(b+1)$ de modo que cada bloco $u_0, \ldots, u_b$ contenha exatamente b elementos. Agora o bloco u_0 contém somente b elementos de modo que ele tem espaço para que inserirmos x.



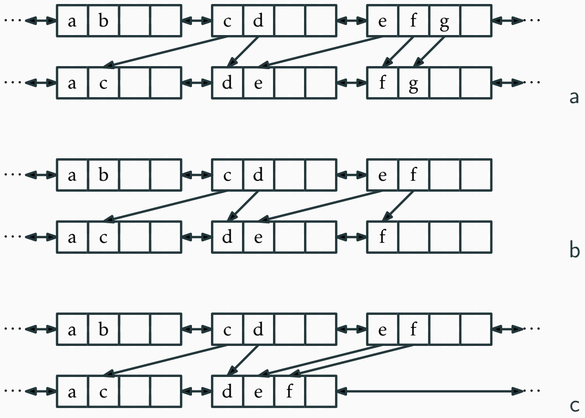
$add(i, x)$

```
add(i, x)
    if i = n then
        append(x)
        return
    u, j ← get_location(i)
    r ← 0
    w ← u
    while r < b and w ≠ dummy and w.d.size() = b + 1 do
        w ← w.next
        r ← r + 1
    if r = b then # b blocos, cada um com b+1 elementos
        espalhar(u)
        w ← u
    if w = dummy then # acabam os blocos - adiciona novo nó
        w ← add_before(w)
    while w ≠ u do # trabalha reverso, deslocando elementos
        w.d.add_first(w.prev.d.remove_last())
        w ← w.prev
    w.d.add(j, x)
    n ← n + 1
```

Removendo um elemento

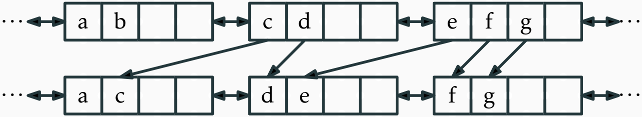
Remover um elemento de uma lista é similiar a adicionar. Primeiro localizamos o nó u que contém o elemento com índice i. Agora, temos que estar preparados para o caso no qual não podemos remover um elemento de u sem tornar o bloco u menor que $b - 1$.

$remove(i)$ - Três casos



$remove(i)$ - Caso a

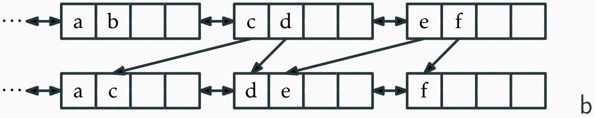
Rapidamente (em $r + 1 \leq b$ passos) encontramos um nó cujo bloco contém mais que $b - 1$ elementos. Neste caso, executamos r deslocamentos de um elemento de um bloco para o anterior, de modo que o elemento extra em u_r se torne um elemento extra em u_0. Agora podemos remover o elemento apropriado do bloco u_0.



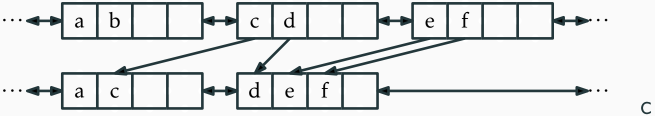
a

$remove(i)$ - Caso b

Rapidamente (em $r + 1 \leq b$ passos) encontramos o fim da lista de blocos. Neste caso, u_r é o último bloco, e não temos necessidade que o bloco u_r possua ao menos $b - 1$ elementos. Assim, prosseguimos com no caso a, emprestando um elemento de u_r para criar um elemento extra em u_0. Se isto provocar que o bloco u_r se torne vazio, o removemos.



Após b passos, não encontramos nenhum bloco contendo mais que $b-1$ elementos. Neste caso, $u_0, \ldots, u_{b-1}$ é uma sequência de b blocos, cada um contendo $b-1$ elementos. Então vamos *concentrar* esses $b(b-1)$ elementos em $u_0, \ldots, u_{b-2}$ de tal modo que cada um desses $b-1$ blocos contenha exatamente b elementos e removemos u_{b-1}, que agora está vazio. Agora o bloco u_0 contém b elementos e nós podemos remover o elemento apropriado dele.



$remove(i)$

```
remove(i)
    u, j ← get_location(i)
    y ← u.d.get(j)
    w ← u
    r ← 0
    while r < b and w ≠ dummy and w.d.size() = b − 1 do
        w ← w.next
        r ← r + 1
    if r = b then # b blocos, cada um com b-1 elementos
        concentrar(u)
    u.d.remove(j)
    while u.d.size() < b − 1 and u.next ≠ dummy do
        u.d.add_last(u.next.d.remove_first())
        u ← u.next
    if u.d.size() = 0 then remove_node(u)
    n ← n − 1
```

espalhar(u)

espalhar(u)

$w \leftarrow u$

for j **in** $0, 1, 2, \ldots, b-1$ **do**

  $w \leftarrow w.next$

$w \leftarrow$ add_before(w)

while $w \neq u$ **do**

  while $w.d$.size() $< b$ **do**

    $w.d$.add_first($w.prev.d$.remove_last())

  $w \leftarrow w.prev$

concentrar(u)
    $w \leftarrow u$
    for j **in** $0, 1, 2, \ldots, b-2$ **do**
        while $w.d.\text{size}() < b$ **do**
            $w.d.\text{add\_last}(w.next.d.\text{remove\_first}())$
        $w \leftarrow w.next$
    remove_node(w)

FIM