Capítulo 3

Listas Encadeadas

Neste capítulo, continuamos a estudar implementações da interface List, desta vez utilizando estruturas de dados baseadas em ponteiro em vez de arrays. As estruturas neste capítulo são constituídas por nós que contêm os itens da lista. Usando referências (ponteiros), os nós são encadeados em uma sequência. Primeiro, estudamos listas simplesmente encadeadas, que podem implementar as operações de uma Stack e de uma Queue (FIFO) em tempo constante por operação e, em seguida, passamos para listas duplamente encadeadas, que podem implementar operações de Deque em tempo constante.

As listas encadeadas têm vantagens e desvantagens quando comparadas com implementações baseadas em array da interface List. A principal desvantagem é que perdemos a capacidade de acessar qualquer elemento usando get(i) ou set(i, x) em tempo constante. Em vez disso, temos de percorrer a lista, um elemento de cada vez, até chegar ao i-ésimo elemento. A principal vantagem é que elas são mais dinâmicas: dada uma referência a qualquer nó de lista u, podemos apagar u ou inserir um nó adjacente a u em tempo constante. Isso é verdade, não importa onde u esteja na lista.

3.1 SLList: Uma Lista Simplesmente Encadeada

Uma SLList (lista simplesmente encadeada) é uma sequência de *Nó*s. 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, $w.next = nil$

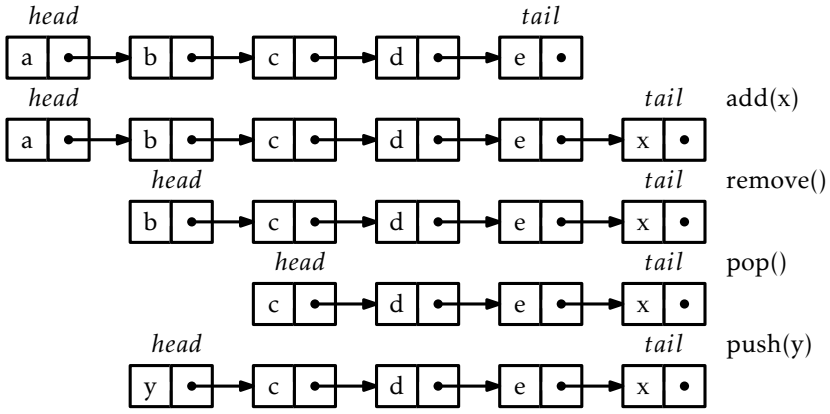

Figura 3.1: Uma sequência de operações de Queue (add(x) e remove()) e de Stack (push(x) e pop()) em uma SLList.

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 ← 0
    head ← nil
    tail ← nil
```

Uma sequência de operações em um Stack e uma Queue em um SLList é ilustrada na Figura 3.1.

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

```
push(x)
    u ← new_node(x)
    u.next ← head
    head ← u
    if n = 0 then
        tail ← u
    n ← n + 1
    return x
```

A operação pop(), depois de verificar que a SLList não está vazia, remove a cabeça definindo $head \leftarrow head.next$ 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 ← head.x
    head ← head.next
    n ← n − 1
    if n = 0 then
        tail ← nil
    return x
```

Claramente, as operações push(x) e pop() são executadas em tempo $O(1)$.

3.1.1 Operações de Fila

Uma SLList também pode implementar as operações de fila FIFO, add(x) e remove(), em tempo constante. As remoções são feitas a partir da cabeça da lista e são idênticas à operação pop():

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

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 $tail = head = nil$. Nesse caso, tanto *tail* como *head* são definidos como u.

```
add(x)
    u ← new_node(x)
    if n = 0 then
        head ← u
    else
        tail.next ← u
    tail ← u
    n ← n + 1
    return true
```

Claramente, ambos $add(x)$ e $remove()$ levam tempo constante.

3.1.2 Resumo

O seguinte teorema resume o desempenho de uma SLList:

Teorema 3.1. *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.

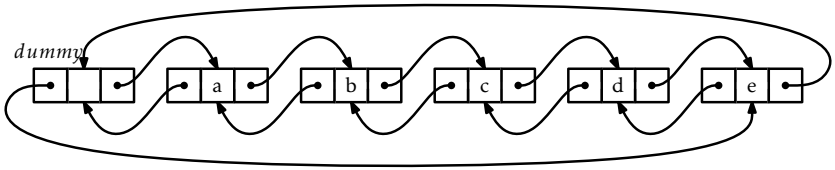

Figura 3.2: Uma DLList contendo a,b,c,d,e.

3.2 DLList: Uma lista duplamente encadeada

A DLList (lista duplamente encadeada) é muito semelhante a uma SLList, 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.

Ao implementar uma SLList, vimos que sempre havia 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 limpa de cuidar de todos esses casos especiais numa DLList é introduzir um nó *dummy*. Este é um nó que não contém quaisquer dados, mas age como um espaço reservado para que não haja nós especiais; cada nó tem um *next* e um *prev*, com o *dummy* agindo como o nó que sucede o último nó na lista e que precede o primeiro nó na lista. Desta forma, os nós da lista são (duplamente) ligados em um ciclo, como ilustrado na Figura 3.2.

```
initialize()
    n ← 0
    dummy ← DLList.Node(nil)
    dummy.prev ← dummy
    dummy.next ← dummy
```

Encontrar um nó com um índice específico em uma DLList é fácil. Podemos começar na cabeça da lista (*dummy.next*) e trabalhar para a frente, ou começar no final da lista (*dummy.prev*) e trabalhar para trás. Isso nos permite alcançar o i-ésimo nó em $O(1 + \min\{i, n - i\})$:

```
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
```

As operações get(i) e set(i,x) agora também são fáceis. Em primeiro lugar, localizamos o i-ésimo nó e depois obtemos(get) ou definimos(set) seu valor x:

```
get(i)
    return get_node(i).x

set(i,x)
    u ← get_node(i)
    y ← u.x
    u.x ← x
    return y
```

O tempo de execução dessas operações é dominado pelo tempo que leva para encontrar o i-ésimo nó, sendo assim, $O(1 + \min\{i, n - i\})$.

3.2.1 Adicionando e Removendo

Se tivermos uma referência a um nó w numa DLList e quisermos inserir um nó u antes de w, então isto é apenas uma questão de definir $u.next = w$, $u.prev = w.prev$ e, em seguida, ajustar $u.prev.next$ e $u.next.prev$. (Ver Figura 3.3.) Graças ao nó dummy, não há necessidade de se preocupar se $w.prev$ ou $w.next$ não existam.

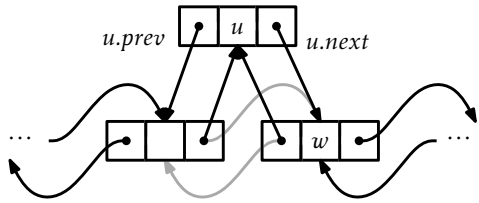

Figura 3.3: Adicionando o nó u antes do nó w em uma DLList.

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

Dessa forma, a operação de lista add(i,x) se torna trivial para implementar. Encontramos o i-ésimo nó na DLList e inserimos um novo nó u que contém x imediatamente antes dele.

```
add(i, x)
    add_before(get_node(i), x)
```

A única parte não constante do tempo de execução de add(i,x) é o tempo necessário para encontrar o i-ésimo nó (usando get_node(i)). Assim, add(i,x) é executado em $O(1+\min\{i, n-i\})$.

Remover um nó w da DLList é fácil. Nós só precisamos ajustar os ponteiros em $w.next$ e $w.prev$ para que eles pulem w. Novamente, o uso do nó dummy elimina a necessidade de considerar quaisquer casos especiais:

```
remove(w)
    w.prev.next ← w.next
    w.next.prev ← w.prev
    n ← n − 1
```

Agora a operação remove(i) é trivial. Achamos o nó com índice i e removemos:

```
remove(i)
    remove(get_node(i))
```

Novamente, a única parte cara dessa operação é achar o i-ésimo nó usando get_node(i), então remove(i) roda em $O(1+\min\{i, n-i\})$.

3.2.2 Resumo

O seguinte teorema resume o desempenho de uma DLList:

Teorema 3.2. *A DLList implementa a interface de uma Lista. Na implementação, as operações* get(i), set(i,x), add(i,x) *e* remove(i) *executam em* $O(1+\min\{i, n-i\})$ *por operação.*

Vale notar que, se ignorarmos o custo da operação get_node(i), então todas as operações da DLList levam tempo constante. Portanto, a única parte cara das operações na DLList é achar o nó relevante. Uma vez tenhamos o nó relevante, adicionar, remover, ou acessar os dados no nó leva tempo constante.

Isso contrasta claramente com as implementações da Lista baseada em array do Capítulo 2; nessas implementações, o item relevante no array pode ser encontrado com tempo constante. Entretanto, adicionar ou remover requer uma mudança de elementos no array e, em geral, leva um tempo não constante.

Por essa razão, as estruturas de lista encadeadas são bem adequadas para aplicações em que as referências a nós de lista podem ser obtidas por meios externos.

3.3 SEList: Uma Lista Encadeada Eficiente em Espaço

Uma das desvantagens das listas encadeadas (além do tempo que leva para acessar elementos que estão no final da lista) é o seu uso de espaço. Cada nó na DLList requer duas referências adicionais para o próximo nó e o anterior na lista. Dois desses campos no nó Node são dedicados a manter a lista, e só um dos campos serve para armazenar dados!

Uma SEList (lista eficiente em espaço) reduz o desperdício de espaço usando uma ideia simples: em vez de guardar os elementos individualmente numa DLList, guardamos um bloco (array) contendo vários itens. Mais precisamente, a SEList é parametrizada pelo *tamanho do bloco b*. Cada nó individual em uma SEList guarda um bloco que suporta até $b+1$ elementos.

Por razões que ficarão claras mais à frente, será útil se pudermos fazer operações do Deque em cada bloco. A estrutura de dados que escolhemos para isso é a BDeque (bounded deque), derivada da ArrayDeque, estrutura descrita na Seção 2.4. O BDeque se diferencia do ArrayDeque numa pequena coisa: quando um novo BDeque é criado, o tamanho do array de suporte a é fixado em $b+1$ e nunca cresce ou encolhe. A propriedade importante do BDeque é que ele permite adição e remoção de elementos por qualquer dos lados em tempo constante. Isso será útil quando elementos forem trocados de um bloco para outro.

Uma SEList é só uma lista duplamente encadeada de blocos. Além dos ponteiros *next* e *prev* , cada nó u em uma SEList contém um BDeque, $u.d$.

3.3.1 Requisitos de Espaço

Uma SEList coloca restrições rígidas sobre o número de elementos em um bloco: a menos que um bloco seja o último bloco, então esse bloco contém pelo menos $b-1$ e no máximo $b+1$ elementos. Isto significa que, se um SEList contém n elementos, então ele tem no máximo

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

blocos. A BDeque para cada bloco contém um array de comprimento $b+1$ mas, para cada bloco exceto o último, no máximo uma quantidade constante de espaço é desperdiçada nesse array. A memória restante usada

por um bloco também é constante. Isso significa que o espaço perdido em uma SEList é apenas $O(b + n/b)$. Ao escolher um valor de b dentro de um fator constante $\sqrt{n}$, podemos fazer o overhead de espaço de uma SEList se aproximar do limite inferior $\sqrt{n}$ dado na Seção 2.6.2.

3.3.2 Encontrando Elementos

O primeiro desafio que encontramos em uma SEList é encontrar o item da lista com um dado índice i. Note que a localização do elemento é constituída por duas partes:

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

Para encontrar o bloco que contém um determinado elemento, procedemos da mesma maneira que em uma DLList. Ou começamos pela frente da lista se deslocando para frente, ou por trás da lista se deslocando para trás até chegar ao nó que queremos. A única diferença é que, cada vez que passamos de um nó para o próximo, nós pulamos um bloco inteiro de elementos.

```
get_location(i)
    if i < n div 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
```

Lembre-se que, com exceção de no máximo um bloco, cada bloco contém pelo menos $b-1$ elementos, de modo que cada etapa em nossa pesquisa nos deixa $b-1$ elementos mais próximos do elemento procurado. Se nós estamos procurando para a frente, isto significa que atingimos o nó procurado após $O(1+i/b)$ passos. Se buscarmos para trás, então alcançamos o nó procurado após $O(1+(n-i)/b)$ passos. O algoritmo leva a menor dessas duas quantidades dependendo do valor de i, então o tempo para localizar o item com o índice i é $O(1+\min\{i,n-i\}/b)$.

Uma vez que saibamos como localizar o item com o índice i, as operações get(i) e set(i,x) traduzem-se em obter ou definir um determinado índice no bloco correto:

```
get(i)
    u, j ← get_location(i)
    return u.d.get(j)

set(i, x)
    u, j ← get_location(i)
    return u.d.set(j, x)
```

Os tempos de execução destas operações são dominados pelo tempo que leva para localizar o item, então eles também são executados no tempo $O(1+\min\{i,n-i\}/b)$.

3.3.3 Adicionando um Elemento

Adicionar elementos em uma SEList é um pouco mais complicado. Antes de considerar o caso geral, consideramos a operação mais fácil, add(x), na qual x é adicionado ao final da lista. Se o último bloco estiver cheio (ou não existir porque ainda não tem blocos), então nós primeiro alocamos um novo bloco e o anexamos à lista de blocos. Agora que temos certeza de que o último bloco existe e não está cheio, anexamos x no último bloco.

```
append(x)
    last ← dummy.prev
```

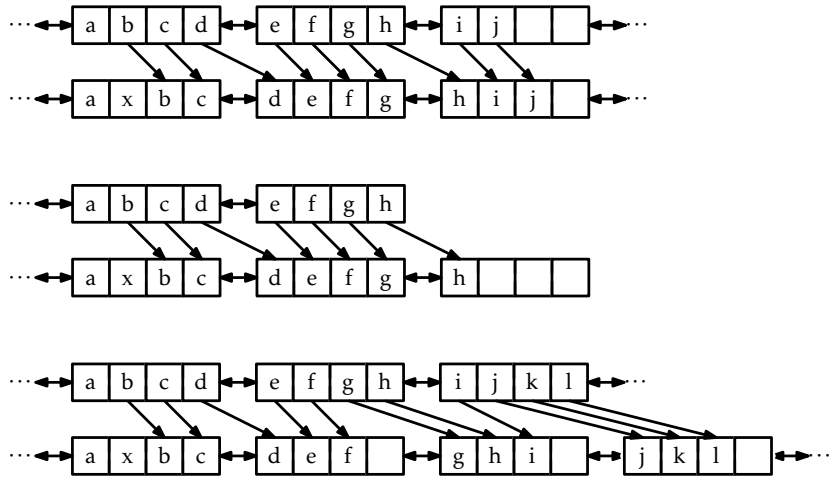

Figura 3.4: Os três casos que ocorrem durante a adição de um item x no interior de uma SEList. (Essa SEList tem bloco de tamanho $b = 3$.)

```
    if last = dummy or last.d.size() = b + 1 then
        last ← add_before(dummy)
    last.d.append(x)
    n ← n + 1
```

As coisas ficam mais complicadas quando adicionamos ao interior da lista usando add(i,x). Primeiro localizamos i para obter o nó u cujo bloco contém o i-ésimo item da lista. O problema é que queremos inserir x no bloco do u, mas temos de estar preparados para o caso onde o bloco do u já contém $b+1$ elementos, já estando cheio e sem espaço para x.

Suponha que $u_0, u_1, u_2, \ldots$ indica u, $u.next$, $u.next.next$, e assim por diante. Exploramos $u_0, u_1, u_2, \ldots$ procurando um nó que pode fornecer espaço para x. Três casos podem ocorrer durante a busca por espaço (veja a Figura 3.4):

1. Nós, rapidamente, (em $r + 1 \leq b$ passos) achamos 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 um espaço

livre em u_r se torne um espaço livre em u_0. Podemos, então, inserir x no bloco u_0.

2. Nós, rapidamente, (em $r+1 \leq b$ passos) chegamos ao fim da lista de blocos. Neste caso, adicionamos um novo bloco vazio ao final da lista de blocos e procedemos como no primeiro caso.

3. Após b passos não encontramos nenhum bloco que não está cheio. Neste caso, $u_0, \ldots, u_{b-1}$ é uma sequência de blocos b, em que cada um contém $b+1$ elementos. Inserimos um novo bloco u_b ao final desta sequência e *distribuímos* os $b(b+1)$ elementos originais para que cada bloco de $u_0, \ldots, u_b$ contenha exatamente b elementos. Agora, o bloco de u_0 contém apenas b elementos, portanto ele tem espaço para inserirmos 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 blocks, each with b+1 elements
        spread(u)
        w ← u
    if w = dummy then # ran off the end - add new node
        w ← add_before(w)
    while w ≠ u do # work backwards, shifting elements as we go
        w.d.add_first(w.prev.d.remove_last())
        w ← w.prev
    w.d.add(j,x)
    n ← n + 1
```

O tempo de execução da operação add(i,x) depende que cada um dos três casos acima ocorra. Os casos 1 e 2 envolvem examinar e deslocar elementos através de, no máximo, b blocos e demoram $O(b)$. O caso 3 envolve chamar o método spread(u), que move $b(b+1)$ elementos e demora $O(b^2)$. Se nós ignorarmos o custo do Caso 3 (o que explicaremos mais adiante com a amortização), isto significa que o tempo de execução total para localizar i e executar a inserção de x é $O(b+\min\{i,n-i\}/b)$.

3.3.4 Remover um Elemento

Remover um elemento de uma SEList é similar a adicionar um elemento. Primeiro, localizamos o nó u que contém o elemento de índice i. Agora, temos que estar preparados para o caso em que não podemos remover um elemento de u sem fazer com que o bloco u fique menor que $b-1$.

Novamente, deixe $u_0, u_1, u_2, \ldots$ indicar u, $u.next$, $u.next.next$, e assim por diante. Examinamos $u_0, u_1, u_2, \ldots$ para procurar um nó do qual podemos pegar emprestado um elemento para fazer o tamanho do bloco u_0 ser, no mínimo, $b-1$. Há três casos a considerar (ver Figura 3.5):

1. Nós, rapidamente, (em $r+1 \leq b$ passos) achamos um nó cujo bloco contém mais de $b-1$ 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. Podemos, então,remover o elemento apropriado do bloco u_0.

2. Nós, rapidamente, (em $r+1 \leq b$ passos) chegamos ao fim da lista de blocos. Neste caso, u_r é o último bloco, e não há razão para o bloco u_r conter, no mínimo, $b-1$ elementos. Portanto, procedemos como acima, pegando emprestado um elemento de u_r para fazer um elemento extra em u_0. Se isto fizer com que o bloco u_r fique vazio, então removemos ele.

3. Após b passos, não encontramos nenhum bloco contendo mais de $b-1$ elementos. Neste caso, $u_0, \ldots, u_{b-1}$ é uma sequência de b blocos, em que cada um contém $b-1$ elementos. Nós *concentramos* estes $b(b-1)$ elementos em $u_0, \ldots, u_{b-2}$ de modo que cada um destes $b-1$ blocos contenha exatamente b elementos e removemos u_{b-1}, que

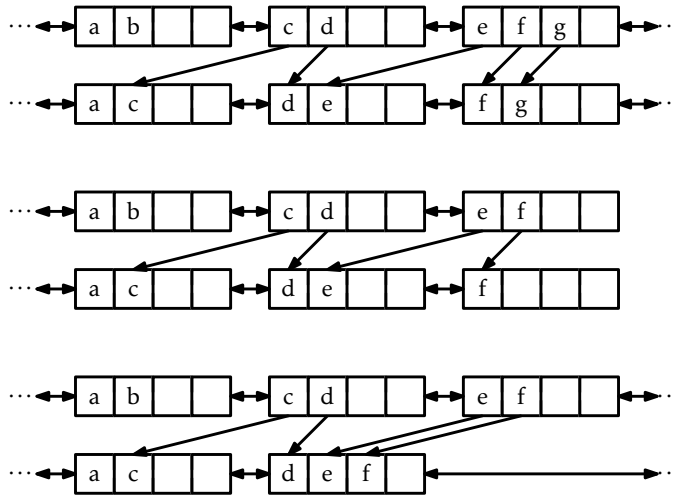

Figura 3.5: Os três casos que ocorrem durante a remoção de um item x no interior de uma SEList. (Esta SEList tem bloco de tamanho $b = 3$.)

agora está vazio. Agora, o bloco de u_0 contém b elementos e, então, podemos remover o elemento apropriado dele.

```
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 blocks, each with b-1 elements
        gather(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
```

Como a operação add(i,x), o tempo de execução da operação remove(i) é $O(b + \min\{i, n-i\}/b)$ se ignorarmos o custo do método gather(u) que ocorre no Caso 3.

3.3.5 Análise Amortizada de Distribuição e Concentração

Em seguida, consideramos o custo dos métodos gather(u) e spread(u) que podem ser executados pelos métodos add(i,x) e remove(i). Por uma questão de completude, aqui estão:

```
spread(u)
    w ← u
    for j in 0, 1, 2, ..., b − 1 do
        w ← w.next
    w ← add_before(w)
    while w ≠ u do
        while w.d.size() < b do
            w.d.add_first(w.prev.d.remove_last())
        w ← w.prev
```

```
gather(u)
    w ← u
    for j in 0, 1, 2, ..., b − 2 do
        while w.d.size() < b do
            w.d.add_last(w.next.d.remove_first())
        w ← w.next
    remove_node(w)
```

O tempo de execução de cada um destes métodos é dominado pelos dois loops aninhados. Ambos os loops, internos e externos, executam no máximo $b+1$ vezes, de modo que o tempo de execução total de cada um

destes métodos é $O((b+1)^2) = O(b^2)$. No entanto, o seguinte lema mostra que estes métodos executam no máximo um de cada b chamadas para add(i,x) ou remove(i).

Lema 3.1. *Se for criado uma SEList vazia e qualquer sequência de $m \geq 1$ chamadas para* add(i,x) *e* remove(i) *for executado, então o tempo total gasto durante as chamadas para* spread() *e* gather() *é $O(bm)$.*

Demonstração. Nós usaremos o método potencial de análise amortizada. Dizemos que um nó u é *frágil* se o bloco u não contém b elementos (de modo que u seja o último nó, ou contenha $b-1$ ou $b+1$ elementos). Qualquer nó cujo bloco contenha b elementos é *rígido*. Defina o *potential* de uma SEList como o número de nós frágeis que contém. Consideramos apenas a operação add(i,x) e sua relação com o número de chamadas para spread(u). A análise de remove(i) e gather(u) é idêntica.

Observe que, se o Caso 1 ocorrer durante o método add(i,x), então somente um nó, u_r, tem o tamanho de seu bloco alterado. Portanto, no máximo um nó, ou seja u_r, irá de rígido a frágil. Se ocorrer o Caso 2, então um novo nó é criado, e este novo nó será frágil, mas nenhum outro nó mudará de tamanho, então o número de nós frágeis aumentará em um. Assim, tanto no Caso 1 ou no Caso 2 o potencial do SEList aumentará para no máximo um.

Finalmente, se ocorre o Caso 3, é porque $u_0,\ldots,u_{b-1}$ são todos nós frágeis. Então $spread(u_0)$ é chamado e estes b nós frágeis são substituídos por $b+1$ nós rígidos. Finalmente, x é adicionado ao bloco u_0, fazendo u_0 frágil. No total, o potencial diminui em $b-1$.

Em resumo, o potencial começa em 0 (não há nós na lista). Cada vez que Caso 1 ou Caso 2 ocorre, o potencial aumenta, em no máximo, 1. Cada vez que ocorre o Caso 3, o potencial diminui para $b-1$. O potencial (que conta o número de nós frágeis) nunca é menor que 0. Nós concluímos que, para cada ocorrência do Caso 3, há pelo menos $b-1$ ocorrências do Caso 1 ou Caso 2. Assim, para cada chamada para spread(u) existem pelo menos b chamadas para add(i,x). Isso completa a verificação. □

3.3.6 Resumo

O seguinte teorema resume o desempenho das estruturas de dados de SEList:

Teorema 3.3. *Um SEList implementa a interface de Lista. Ignorando o custo das chamadas para* spread(u) *e* gather(u)*, uma SEList com tamanho de bloco b supporta as operações*

- get(i) *e* set(i,x) *em* $O(1+\min\{i, n-i\}/b)$ *vezes por operação; e*
- add(i,x) *e* remove(i) *em* $O(b+\min\{i, n-i\}/b)$ *vezes por operação.*

Além disso, começando com uma SEList vazia, qualquer sequência de operações m add(i,x) *e* remove(i) *resulta em um tempo total gasto de $O(bm)$ durante todas as chamadas para* spread(u) *e* gather(u)*.*

O espaço (medido em palavras)[1] *usado por uma SEList que armazena n elementos é $n + O(b + n/b)$.*

A SEList é um compromisso entre uma ArrayList e uma DLList na qual a mistura relativa destas duas estruturas depende do tamanho do bloco b. No extremo $b = 2$, cada SEList armazena no máximo três valores, o que não é muito diferente da DLList. No outro extremo, $b > n$, todos os elementos são armazenados em um único array, assim como em um ArrayList. Entre esses dois extremos reside uma troca entre o tempo que leva para adicionar ou remover um item da lista e o tempo que leva para localizar um item de lista particular.

3.4 Discussão e Exercícios

Tanto as listas simplesmente encadeadas e as listas duplamente encadeadas são técnicas estabelecidas, tendo sido utilizadas em programas há mais de 40 anos. Elas são discutidas, por exemplo, por Knuth [46, Seções 2.2.3–2.2.5]. Mesmo a estrutura de dados SEList parece ser um exercício bem conhecido de estruturas de dados. A SEList é as vezes chamada de *unrolled linked list* [67].

[1] Releia a Seção 1.4 para uma discussão de como a memória é medida.

Outra maneira de economizar espaço em uma lista duplamente encadeada é usar as chamadas listas XOR. Em uma lista XOR, cada nó, u, contém apenas um ponteiro, chamado $u.nextprev$, que contém o *ou exclusivo bi-a-bit* de $u.prev$ e $u.next$. A própria lista precisa armazenar dois ponteiros, um para o nó *dummy* e um para $dummy.next$ (o primeiro nó, ou *dummy* se a lista estiver vazia). Esta técnica utiliza o fato de que, se temos um ponteiro para u e $u.prev$, então podemos extrair $u.next$ usando a fórmula

$$u.next = u.prev \hat{} u.nextprev \ .$$

(Aqui $\hat{}$ calcula a operação bit-a-bit ou-exclusivo de seus dois argumentos.) Esta técnica complica um pouco o código e não é possível em algumas linguagens como Java e Python, que têm coleta de lixo, porém fornece uma lista duplamente encadeada que necessita apenas de um ponteiro por nó, Veja o artigo de Sinha [68] para uma discussão detalhada para listas XOR.

Exercício 3.1. Por que não é possível usar um nó dummy num SLList para evitar todos os casos especiais que ocorrem nas operações push(x), pop(), add(x) e remove()?

Exercício 3.2. Projete e implemente um método para a SLList, second_last(), que retorna o penúltimo elemento de SLList. Faça isso sem usar a variável de membro, n, que acompanha o tamanho da lista.

Exercício 3.3. Implementar as operações de List, get(i), set(i,x), add(i,x) e remove(i) em SLList. Cada uma dessas operações deve ser executada em um tempo $O(1+i)$.

Exercício 3.4. Projete e implemente um método para a SLList, reverse() que inverte a ordem dos elementos em SLList. Este método deve ser executado em tempo $O(n)$, não deve usar recursão, não deve usar nenhuma estrutura de dados secundária e não deve criar novos nós.

Exercício 3.5. Projete e implemente os métodos para a SLList e DLList chamados check_size(). Esses métodos percorrem a lista e contam o número de nós para ver se isso corresponde ao valor, n, armazenado na lista. Esses métodos não retornam nada, mas lançam uma exceção se o tamanho que eles compõem não corresponde ao valor de n.

Exercício 3.6. Tente recriar o código para a operação add_before(w) que cria um nó, u, e adiciona-o em DLList antes do nó w. Não consulte este capítulo. Mesmo que seu código não coincida exatamente com o código fornecido neste livro, ele ainda pode estar correto. Teste e veja se ele funciona.

Os próximos exercícios envolvem a realização de manipulações em DLList. Você deve completá-los sem alocar novos nós ou matrizes temporárias. Todos podem ser feitos apenas alterando os valores *prev* e *next* dos nós existentes.

Exercício 3.7. Escreva um método is_palindrome() para uma DLList que retorna *true* se a lista for *palíndromo*, ie, o elemento na posição i é igual ao elemento na posição $n-i-1$ para todos $i \in \{0,\ldots,n-1\}$. Seu código deve ser executado em tempo $O(n)$.

Exercício 3.8. Implementar um método rotate(r) que "rotaciona" uma DLList para que o item da lista i se torne o item da lista $(i+r) \bmod n$. Esse método deve ser executado em tempo $O(1+\min\{r, n-r\})$ e não deve modificar nenhum nó na lista.

Exercício 3.9. Escreva um método, truncate(i), que trunca uma DLList na posição i. Depois de executar este método, o tamanho da lista será i e deve conter apenas os elementos nos índices $0,\ldots,i-1$.. O valor de retorno é outra DLList que contém os elementos nos índices $i,\ldots,n-1$. Esse método deve ser executado em tempo $O(\min\{i, n-i\})$.

Exercício 3.10. Escreva um método absorb(l_2) para uma DLList, que leva como argumento uma DLList, l_2, esvazia-o e acrescenta seu conteúdo, em ordem, ao receptor. Por exemplo, se l_1 contiver a,b,c e l_2 contém d,e,f, e depois de chamar l_1.absorb(l_2), l_1 conterá a,b,c,d,e,f e l_2 estará vazia.

Exercício 3.11. Escreva um método deal() que remove todos os elementos com índices de números ímpares da DLList e retorna uma DLList contendo esses elementos. Por exemplo, se l_1 contém os elementos a,b,c,d,e,f, depois de chamar l_1.deal(), l_1 deve conter a,c,e e uma lista contendo b,d,f deve ser devolvida.

Exercício 3.12. Escreva um método, reverse(), que inverta a ordem dos elementos em uma DLList.

Exercício 3.13. Este exercício orienta você através de uma implementação do algoritmo merge-sort para classificar uma DLList, como discutido na Seção 11.1.1.

1. Escreva o método DLList chamado take_first(l_2). Este método leva o primeiro nó de l_2 e adiciona-o à lista de recepção. Isso equivale a add(size(),l_2.remove(0)), exceto que ele não deve criar um novo nó.

2. Escreva um método estático de DLList, merge(l_1,l_2), que recebe duas listas classificadas l_1 e l_2, mescla-as e retorna uma nova lista contendo o resultado. Isso tem como consequência que l_1 e l_2 se tornam vazias no processo. Por exemplo, se l_1 contém a, c, d e l_2 contém b, e, f, assim este método retorna uma nova lista contendo a, b, c, d, e, f.

3. Escreva um método DLList sort() que classifica os elementos contidos na lista usando o algoritmo de ordenação por mesclagem. Este algoritmo recursivo funciona da seguinte maneira :

   (a) Se a lista contém 0 ou 1 elementos, então não há nada a fazer. Caso contrário,

   (b) Usando o método truncate(size()/2), divide a lista em duas listas de comprimento aproximadamente igual, l_1 e l_2;

   (c) Recursivamente classificar l_1;

   (d) Recursivamente classificar l_2; e, finalmente,

   (e) Mesclar l_1 e l_2 em uma única lista ordenada.

Os próximos exercícios são mais avançados e requerem uma compreensão do que acontece com o valor mínimo armazenado em uma Stack ou Queue à medida que os itens são adicionados e removidos.

Exercício 3.14. Projetar e implementar uma estrutura de dados MinStack que pode armazenar elementos comparáveis e suporta as operações de pilha push(x), pop(), e size(), assim como a operação de min(), que retorna o valor mínimo atualmente armazenado na estrutura de dados. Todas as operações devem ser executadas em tempo constante.

Exercício 3.15. Projetar e implementar uma estrutura de dados MinQueue que pode armazenar elementos comparáveis e suporta as operações de fila add(x), remove(), e size(), assim como a operação de min(),

que retorna o valor mínimo atualmente armazenado na estrutura de dados. Todas as operações devem ser executadas em tempo constante.

Exercício 3.16. Projetar e implementar a MinDeque uma estrutura de dados que pode armazenar elementos comparáveis e suporta todas as operações de deque add_first(x), add_last(x) remove_first(), remove_last() e size(),e a operação min(), que retorna o menor valor atualmente armazenado em estrutura de dados. Todas as operações devem ser executadas em tempo constante.

Os próximos exercícios são projetados para testar o entendimento do leitor da implementação e análise da lista eficiente em espaço SEList:

Exercício 3.17. Prove que, se uma SEList é usada como uma Stack (para que as únicas modificações no SEList sejam feitas usando push(x) ≡ add(size(), x) e pop() ≡ remove(size() − 1)), então esta operação executado em tempo amortizado constante, independente do valor de b.

Exercício 3.18. Projetar e implementar uma versão de um SEList que suporte todas as operações Deque em tempo amortizado constante por operação, independente do valor de b.

Exercício 3.19. Explicar como usar o operador exclusivo de bit ou operador, ^, para trocar os valores de duas variáveis *int* sem usar uma terceira variável.