Capítulo 4

Skiplists

Neste capítulo discutiremos uma bela estrutura de dados: a skiplist, que possui diversas de aplicações. Usando uma skiplist, podemos implementar uma Lista que tenha tempo de $O(\log n)$ para implementações de $\mathrm{get}(i)$, $\mathrm{set}(i,x)$, $\mathrm{add}(i,x)$, e $\mathrm{remove}(i)$. Nós também podemos implementar uma SSet em que todas as operações são executadas com tempo esperado de $O(\log n)$.

A eficiência das skiplists se baseia no uso da randomização. Quando um novo elemento é adicionado à skiplist, ela utiliza lançamentos aleatórios de moeda para determinar a altura do novo elemento. O desempenho das skiplists é expresso em termos do tempo de execução esperado e do tamanho do caminho. Esta expectativa é baseada nos lançamentos aleatórios de moeda usados pela skiplist. Na implementação, os lançamentos aleatórios de moedas usados pela skiplist são simulados usando um gerador de números (ou bits) pseudo aleatórios.

4.1 A Estrutura Básica

Conceitualmente, uma skiplist é uma sequência de listas simplesmente encadeadas $L_0,\dots,L_h$. Cada lista L_r contém um subconjunto de itens em L_{r-1}. Começamos com a lista inicial L_0 que contém n itens e construímos L_1 a partir de L_0, L_2 a partir de L_1, e assim por diante.

Os itens em L_r são obtidos lançando a moeda para cada elemento, x, em L_{r-1} e incluindo x em L_r se a moeda der cara. Este processo termina

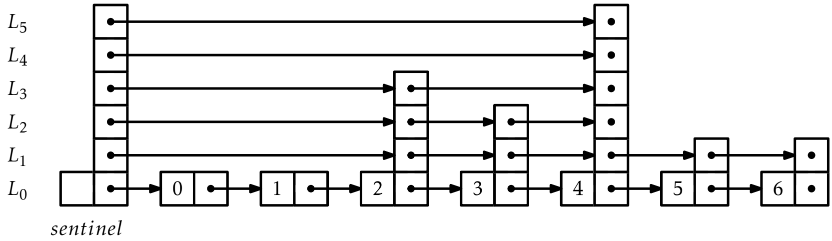

Figura 4.1: Uma skiplist com sete elementos.

quando criamos uma lista L_r que está vazia. Um exemplo de uma skiplist é mostrado na Figura 4.1.

Para um elemento, x, na skiplist, nós chamamos de *altura* de x o valor mais alto de r de modo que x apareça em L_r. Assim, por exemplo, elementos que só aparecem em L_0 tem altura 0. Se pararmos um momento pra pensar sobre isso, percebemos que a altura de x corresponde à seguinte experiência: lance uma moeda repetidamente até aparecer como coroa. Quantas vezes apareceu cara? A resposta, sem surpresa, é que a altura esperada de um nó é 1. (Esperamos lançar a moeda duas vezes antes de aparecer coroa, mas não contamos a última jogada.) A *altura* de uma skiplist é a altura do seu nó mais alto.

No começo de cada lista está um nó especial, chamado de *sentinela*, que atua como um pseudo nó (*dummy*) para a lista. A propriedade chave das skiplists é que existe um caminho curto para busca, chamado de *caminho de busca*, do sentinela em L_h para cada nó em L_0. Veja como é fácil construir um caminho de busca para o nó u (veja Figura 4.2): comece no canto superior esquerdo da skiplist (o sentinela em L_h) e sempre vá para a direita, a menos que ultrapasse u, nesse caso você deve dar um passo para a lista de baixo.

Mais precisamente, para construir um caminho de busca para o nó u em L_0, começaremos pelo sentinela, w, em L_h. Em seguida, verificamos $w.next$. Se $w.next$ contém um elemento que aparece antes de u em L_0, então nós ajustamos $w = w.next$. Caso contrário, movemos para baixo e continuamos a busca da ocorrência de w na lista L_{h-1}. Continuamos assim até chegar ao antecessor de u em L_0.

O resultado a seguir, que vamos provar em Seção 4.4, mostra que o

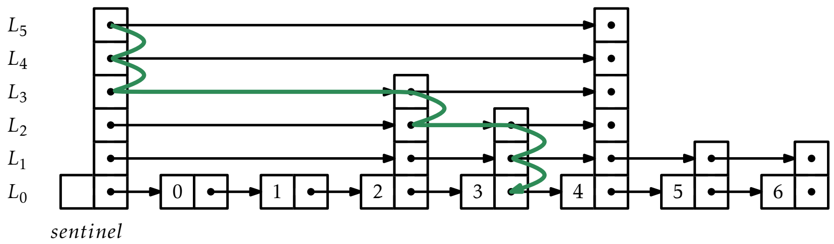

Figura 4.2: Caminho de busca para o nó que contém o valor 4 em uma skiplist.

caminho de busca é bastante curto:

Lema 4.1. *O comprimento esperado do caminho de busca para qualquer nó, u, em* L_0 *é de no máximo* $2\log n + O(1) = O(\log n)$.

Uma maneira eficiente em termos de espaço para implementar uma skiplist é definir um Node, *u*, consistindo em um valor de dados, *x* e um array, *next*, de ponteiros, onde *u.next*[*i*] aponta para o sucessor de *u* na lista L_i. Desta forma, os dados, *x*, em um nó são referenciados apenas uma vez, embora *x* possa aparecer em várias listas.

As próximas duas seções deste capítulo abordam duas aplicações diferentes de skiplists. Em cada uma dessas aplicações, L_0 armazena a estrutura principal (uma lista de elementos ou um conjunto de elementos ordenados). A principal diferença entre essas estruturas é a forma como um caminho de busca é navegado; em particular, elas se diferem em como é decidido se um caminho de busca deve descer em L_{r-1} ou ir para a direita dentro de L_r.

4.2 SkiplistSSet: Uma SSet eficiente

Uma SkiplistSSet usa uma skiplist para implementar a interface SSet. Quando usada desta maneira, a lista L_0 armazena os elementos de SSet de forma ordenada. O método find(x) funciona seguindo o caminho de busca para o menor valor y tal que $y \geq x$:

```
find_pred_node(x)
    u ← sentinel
    r ← h
    while r ≥ 0 do
        while u.next[r] ≠ nil and u.next[r].x < x do
            u ← u.next[r] # go right in list r
        r ← r − 1 # go down into list r-1
    return u

find(x)
    u ← find_pred_node(x)
    if u.next[0] = nil then return nil
    return u.next[0].x
```

Seguir o caminho de busca para y é fácil. Quando situado em algum nó, u, em L_r, olhamos diretamente para $u.next[r].x$, se $x > u.next[r].x$, então damos um passo à direita em L_r. Caso contrário, descemos para L_{r-1}. Cada passo (para direita ou descendo) nesta pesquisa leva um tempo constante; assim, para Lema 4.1, o tempo de execução esperado de find(x) é de $O(\log n)$.

Antes de poder adicionar um elemento a SkipListSSet, precisamos de um método para simular o lançamento de moedas que determina a altura, k, de um novo nó. Fazemos isso escolhendo um número inteiro aleatório, z, e contando o número de 1s na representação binária de z: [1]

```
pick_height()
    z ← random.getrandbits(32)
    k ← 0
    while z ∧ 1 do
        k ← k + 1
        z ← z div 2
```

[1] Este método não reproduz exatamente a experiência de lançar moedas, uma vez que o valor de k será sempre inferior ao número de bits em um *int*. Contudo, isso terá um impacto insignificante, a menos que o número de elementos na estrutura seja muito maior do que $2^{32} = 4294967296$.

```
    return k
```

Para implementar o método add(x) na SkiplistSSet procuramos x e então colocamos x em algumas listas $L_0, \ldots, L_k$, onde k é selecionado usando o método pick_height(). A maneira mais fácil de fazer isso é usar um array, *stack*, que acompanha os nós em que o caminho de busca desce de alguma lista L_r para L_{r-1}. Mais precisamente, $stack[r]$ é o nó em L_r onde o caminho de busca prosseguiu para L_{r-1}. Os nós que modificamos para inserir x são precisamente os nós $stack[0], \ldots, stack[k]$. O código seguinte implementa este algoritmo para add(x):

```
add(x)
    u ← sentinel
    r ← h
    while r ≥ 0 do
        while u.next[r] ≠ nil and u.next[r].x < x do
            u ← u.next[r]
        if u.next[r] ≠ nil and u.next[r].x = x then return false
        stack[r] ← u
        r ← r − 1
    w ← new_node(x, pick_height())
    while h < w.height() do
        h ← h + 1
        stack[h] ← sentinel # height increased
    for i in 0, 1, 2, ..., len(w.next) − 1 do
        w.next[i] ← stack[i].next[i]
        stack[i].next[i] ← w
    n ← n + 1
    return true
```

A remoção de um elemento, x, é feita de forma semelhante, exceto que não há necessidade do *stack* para manter o controle do caminho de busca. A remoção pode ser feita enquanto seguimos o caminho de busca. Procuramos por x e cada vez que a pesquisa se move para baixo a partir do nó u, verificamos se $u.next.x = x$ e, em caso afirmativo, desligamos u

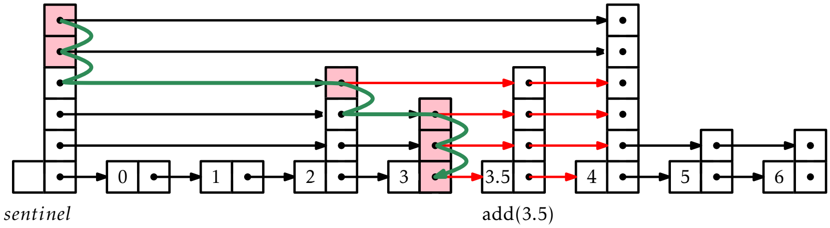

Figura 4.3: Adicionando o nó contendo o valor 3.5 a uma skiplist. Os nós armazenados em *stack* estão marcados.

da lista:

```
remove(x)
  removed ← false
  u ← sentinel
  r ← h
  while r ≥ 0 do
    while u.next[r] ≠ nil and u.next[r].x < x do
      u ← u.next[r]
    if u.next[r] ≠ nil and u.next[r].x = x then
      removed ← true
      u.next[r] ← u.next[r].next[r]
      if u = sentinel and u.next[r] = nil then
        h ← h − 1 # height has decreased
    r ← r − 1
  if removed then n ← n − 1
  return removed
```

4.2.1 Resumo

O seguinte teorema resume o desempenho de uma skiplists quando usada para implementar conjuntos ordenados:

Teorema 4.1. *SkiplistSSet implementa a interface SSet. Uma SkiplistSSet*

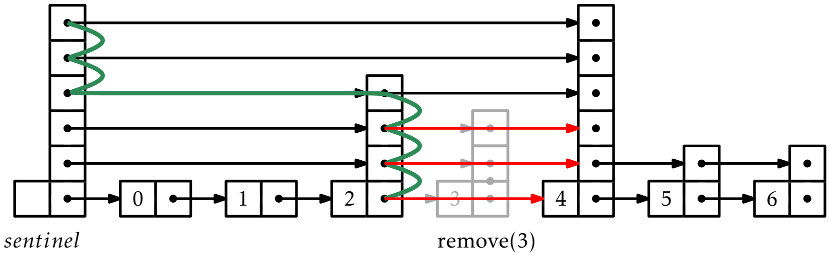

Figura 4.4: Removendo o nó que contém o valor 3 da skiplist.

suporta as operações add(x), remove(x), *e* find(x) *em um tempo esperado* $O(\log n)$ *por operação.*

4.3 SkiplistList: Uma Lista de acesso aleatório eficiente

Uma SkiplistList implementa a interface List usando uma estrutura skiplist. Em uma SkiplistList, L_0 contém os elementos da lista na ordem em que aparecem na lista. Como em uma SkiplistSSet, os elementos podem ser adicionados, removidos, e acessados em um tempo $O(\log n)$.

Para que isso seja possível, precisamos de uma maneira para seguir o caminho de busca para o *i*ésimo elemento em L_0. A maneira mais fácil de fazer isso é definir o conceito de *comprimento* de uma aresta em uma lista, L_r. Definimos o tamanho de cada aresta em L_0 como 1. O tamanho de uma aresta, e, em L_r, $r > 0$, é definido como a soma dos tamanhos das arestas abaixo de e em L_{r-1}. De forma equivalente, o tamanho de e é o número de arestas de L_0 abaixo de e. Veja Figura 4.5 para um exemplo de uma skiplist mostrando o tamanho de suas arestas. Uma vez que as arestas das skiplists são armazenados em arrays, os tamanhos podem ser armazenados da mesma maneira.

A propriedade útil desta definição de tamanho é que, se estamos em um nó na posição j em L_0 e seguimos uma aresta de tamanho ℓ, então movemos para um nó cuja posição, em L_0, é $j + \ell$. Desta forma, enquanto seguimos um caminho de busca, podemos manter o controle da posição, j, do nó atual em L_0. Em um nó, u, em L_r, vamos para a direita se j mais

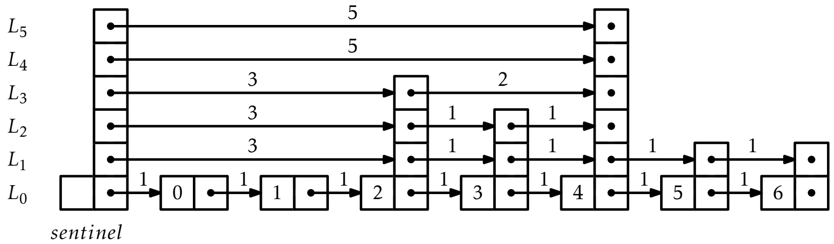

Figura 4.5: Os tamanhos das arestas em uma skiplist.

o tamanho de $u.next[r]$ é menor que i. Caso contrário, desceremos para L_{r-1}.

find_pred(i)
  $u \leftarrow sentinel$
  $r \leftarrow h$
  $j \leftarrow -1$
  while $r \geq 0$ **do**
    while $u.next[r] \neq nil$ **and** $j + u.length[r] < i$ **do**
      $j \leftarrow j + u.length[r]$
      $u \leftarrow u.next[r]$ # go right in list r
    $r \leftarrow r - 1$ # go down into list r-1
  return u

get(i)
  return find_pred(i).$next[0].x$

set(i, x)
  $u \leftarrow$ find_pred(i).$next[0]$
  $y \leftarrow u.x$
  $u.x \leftarrow x$
  return y

Como a parte mais difícil das operações get(i) e set(i, x) é encontrar o

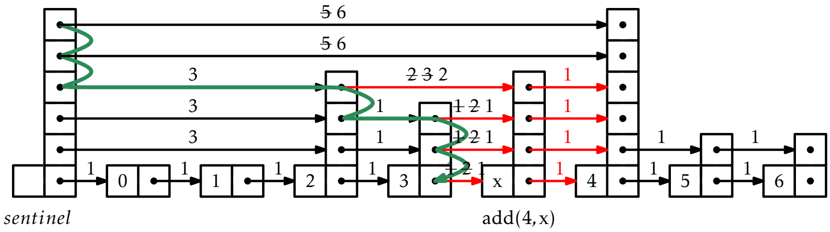

Figura 4.6: Adicionando um elemento em uma SkiplistList.

i-ésimo nó em L_0, essas operações são executadas em um tempo $O(\log n)$.

Adicionar um elemento a uma SkiplistList em uma posição, i, é relativamente simples. Ao contrário do que acontece em uma SkiplistSSet, temos certeza que um novo nó será realmente adicionado, assim podemos realizar a adição ao mesmo tempo em que buscamos a localização do novo nó. Primeiramente, pegamos a altura, k, do nó recentemente inserido, w, e então avançamos o caminho de busca para i. Toda vez que o caminho de busca desce de L_r com $r \leq k$, juntamos w em L_r. O único cuidado extra necessário é assegurar que o comprimento das arestas seja atualizado devidamente. Veja Figura 4.6.

Note que, cada vez que o caminho de busca desce um nó, u, em L_r, o comprimento da aresta $u.next[r]$ aumenta em um, uma vez que estamos adicionando um elemento abaixo desta aresta na posição i. Unir o nó w entre dois nós, u e z, funciona como mostrado na Figura 4.7. Enquanto seguimos o caminho de busca, já estamos cientes da posiçao, j, de u em L_0. Portanto, sabemos que o comprimento da aresta de u a w é $i-j$. Também podemos deduzir o comprimento da aresta de w a z através do comprimento, ℓ, da aresta de u a z. Assim sendo, podemos juntar em w e atualizar os comprimentos das arestas em tempo constante.

Isto soa mais complicado do que é, o código, na verdade, é bem simples:

add(i,x)
  $w \leftarrow$ new_node(x, pick_height())
  if w.height() $> h$ **then**
    $h \leftarrow w$.height()

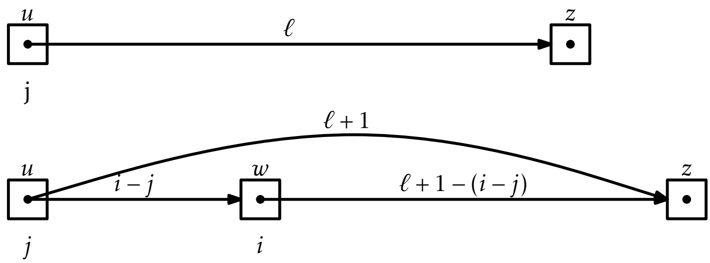

Figura 4.7: Atualizando os comprimentos das arestas enquanto junta-se o nó w em uma skiplist.

add(i,w)

---

add(i,w)
  $u \leftarrow sentinel$
  $k \leftarrow w$.height()
  $r \leftarrow h$
  $j \leftarrow -1$
  while $r \geq 0$ **do**
    while $u.next[r] \neq nil$ **and** $j + u.length[r] < i$ **do**
      $j \leftarrow j + u.length[r]$
      $u \leftarrow u.next[r]$
    $u.length[r] \leftarrow u.length[r] + 1$
    if $r \leq k$ **then**
      $w.next[r] \leftarrow u.next[r]$
      $u.next[r] \leftarrow w$
      $w.length[r] \leftarrow u.length[r] - (i - j)$
      $u.length[r] \leftarrow i - j$
    $r \leftarrow r - 1$
  $n \leftarrow n + 1$
  return u

Agora a implementação da operação remove(i) em uma SkiplistList deveria ser óbvia. Seguimos o caminho de busca para o nó na posição i.

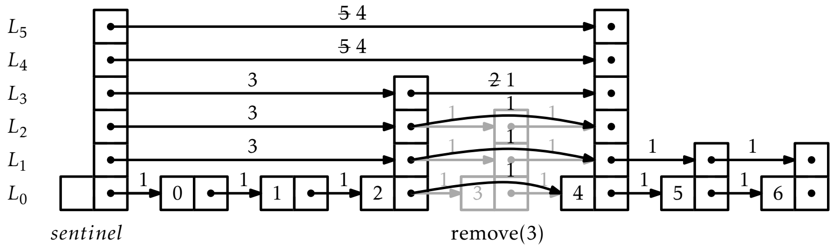

Figura 4.8: Removing an element from a SkiplistList.

Cada vez que o caminho de busca desce de um nó, u, no nível r diminuímos o comprimento da aresta deixando u neste nível. Também checamos se $u.next[r]$ é o elemento de posição i e, caso seja, o tiramos da lista neste nível. Um exemplo é mostrado na Figura 4.8.

remove(i)
    $u \leftarrow sentinel$
    $r \leftarrow h$
    $j \leftarrow -1$
    while $r \geq 0$ **do**
        while $u.next[r] \neq nil$ **and** $j + u.length[r] < i$ **do**
            $j \leftarrow j + u.length[r]$
            $u \leftarrow u.next[r]$
        $u.length[r] \leftarrow u.length[r] - 1$
        if $j + u.length[r] + 1 = i$ **and** $u.next[r] \neq nil$ **then**
            $x \leftarrow u.next[r].x$
            $u.length[r] \leftarrow u.length[r] + u.next[r].length[r]$
            $u.next[r] \leftarrow u.next[r].next[r]$
            if $u = sentinel$ **and** $u.next[r] = nil$ **then**
                $h \leftarrow h - 1$
        $r \leftarrow r - 1$
    $n \leftarrow n - 1$
    return x

4.3.1 Resumo

O teorema a seguir resume o desempenho da estrutura de dados SkiplistList:

Teorema 4.2. *Uma SkiplistList implementa a interface List. Uma SkiplistList suporta as operações* get(i), set(i,x), add(i,x), *e* remove(i) *no tempo de execução $O(\log n)$ esperado.*

4.4 Análise de Skiplists

Nesta seção, analisamos a altura, tamanho e comprimento esperados do caminho de busca em uma skiplist. Esta seção requer um conhecimento de probabilidade básica. Várias provas são baseadas nas seguintes observações básicas sobre lançamentos de moedas.

Lema 4.2. *Faça com que T seja o número de vezes que uma moeda é jogada e incluindo a primeira vez que a moeda dá cara. Logo* $\mathrm{E}[T] = 2$.

Demonstração. Suponhamos que paremos de jogar a moeda na primeira vez que a moeda der cara. Define-se a variável indicadora

$$I_i = \begin{cases} 0 & \text{se a moeda for lançada menos que } i \text{ vezes} \\ 1 & \text{se a moeda for lançada } i \text{ ou mais vezes} \end{cases}$$

Note que $I_i = 1$ se e apenas se, os primeiros $i-1$ lançamentos da moeda derem coroa, então $\mathrm{E}[I_i] = \Pr\{I_i = 1\} = 1/2^{i-1}$. Observe que T, o número total de lançamentos da moeda, pode ser escrito como $T = \sum_{i=1}^{\infty} I_i$. Portanto,

$$\begin{aligned} \mathrm{E}[T] &= \mathrm{E}\left[\sum_{i=1}^{\infty} I_i\right] \\ &= \sum_{i=1}^{\infty} \mathrm{E}[I_i] \\ &= \sum_{i=1}^{\infty} 1/2^{i-1} \\ &= 1 + 1/2 + 1/4 + 1/8 + \cdots \\ &= 2 \ . \end{aligned}$$

□

Os dois próximos lemas nos dizem que as skiplists têm tamanho linear:

Lema 4.3. *O número esperado de nós em uma skiplist contendo n elementos, não incluindo ocorrências do sentinela, é $2n$.*

Demonstração. A probabilidade de que qualquer elemento particular, x, esteja incluso na lista L_r é $1/2^r$, então o número de nós esperado na L_r é $n/2^r$.[2] Portanto, o número total de nós esperado em todas as listas é

$$\sum_{r=0}^{\infty} n/2^r = n(1+1/2+1/4+1/8+\cdots) = 2n \ . \qquad \square$$

Lema 4.4. *A altura esperada de uma skiplist contendo n elements é no máximo* $\log n + 2$.

Demonstração. Para cada $r \in \{1,2,3,\ldots,\infty\}$, defina a variável indicadora aleatória

$$I_r = \begin{cases} 0 & \text{se } L_r \text{ é vazia} \\ 1 & \text{se } L_r \text{ não é vazia} \end{cases}$$

A altura, h, da skiplist é dada por

$$h = \sum_{r=1}^{\infty} I_r \ .$$

Note que I_r nunca é maior que o tamanho, $|L_r|$, de L_r, assim

$$\mathrm{E}[I_r] \le \mathrm{E}[|L_r|] = n/2^r \ .$$

[2] Veja Seção 1.3.4 para ver como isso é derivado usando variáveis indicadoras e linearidade de expectativa.

Portanto, teremos

$$
\begin{aligned}
\mathrm{E}[h] &= \mathrm{E}\left[\sum_{r=1}^{\infty} I_r\right] \\
&= \sum_{r=1}^{\infty} E[I_r] \\
&= \sum_{r=1}^{\lfloor \log n \rfloor} E[I_r] + \sum_{r=\lfloor \log n \rfloor+1}^{\infty} E[I_r] \\
&\leq \sum_{r=1}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor+1}^{\infty} n/2^r \\
&\leq \log n + \sum_{r=0}^{\infty} 1/2^r \\
&= \log n + 2 \ .
\end{aligned}
$$

□

Lema 4.5. *O número esperado de nós em uma skiplist contendo n elementos, incluindo todas as ocorrências do sentinela, é* $2n + O(\log n)$.

Demonstração. Por Lema 4.3, o número esperado de nós, sem incluir o sentinela, é $2n$. O número de ocorrências do sentinela é igual à altura, h, da skiplist assim, por Lema 4.4 o número esperado de ocorrências do sentinela é no máximo $\log n + 2 = O(\log n)$. □

Lema 4.6. *O comprimento esperado do caminho de busca em uma skiplist é no máximo* $2\log n + O(1)$.

Demonstração. A maneira mais fácil de ver isso é considerar o *caminho de pesquisa reversa* para um nó, x. Este caminho começa no predecessor de x em L_0. A qualquer momento, se o caminho puder subir um nível, isso acontecerá. Se não conseguir subir um nível, ele irá para a esquerda. Pensar nisso por alguns instantes nos convencerá de que o caminho de pesquisa reversa para x é idêntico ao caminho de pesquisa para x, exceto que é invertido.

O número de nós que o caminho de pesquisa reverso visita em um nível específico, r, está relacionado à seguinte experiência: lançar uma moeda. Se a moeda surgir como cara, vá para cima e pare. Caso contrário, vá para a esquerda e repita a experiência. O número de lançamentos antes da cara representa o número de passos à esquerda que um caminho de busca reversa toma em um determinado nível. [3] Lema 4.2 nos diz que o número esperado de lançamentos de moeda antes das primeiras caras é 1.

Seja S_r o número de passos que o caminho de busca direta no nível r que vão para a direita. Acabamos de argumentar que $\mathrm{E}[S_r] \leq 1$. Além disso, $S_r \leq |L_r|$, já que não podemos dar mais passos em L_r do que o tamanho de L_r, assim

$$\mathrm{E}[S_r] \leq \mathrm{E}[|L_r|] = n/2^r \enspace .$$

Podemos agora terminar como na prova de Lema 4.4. Seja S o comprimento do caminho de busca para algum nó, u, em uma skiplist, e seja h a altura da skiplist. Então

$$\begin{aligned}
\mathrm{E}[S] &= \mathrm{E}\left[h + \sum_{r=0}^{\infty} S_r\right] \\
&= \mathrm{E}[h] + \sum_{r=0}^{\infty} \mathrm{E}[S_r] \\
&= \mathrm{E}[h] + \sum_{r=0}^{\lfloor \log n \rfloor} \mathrm{E}[S_r] + \sum_{r=\lfloor \log n \rfloor+1}^{\infty} \mathrm{E}[S_r] \\
&\leq \mathrm{E}[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor+1}^{\infty} n/2^r \\
&\leq \mathrm{E}[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r \\
&\leq \mathrm{E}[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r \\
&\leq \mathrm{E}[h] + \log n + 3 \\
&\leq 2\log n + 5 \enspace .
\end{aligned}$$

□

[3] Observe que isso pode sobrecarregar o número de etapas à esquerda, já que a experiência deve terminar em as primeiras caras ou quando o caminho de busca alcança o sentinela, o que ocorrer primeiro. Isto não é um problema já que o lema está apenas indicando um limite superior.

O seguinte teorema resume os resultados desta seção:

Teorema 4.3. *Uma skiplist contendo n elementos tem tamanho esperado* $O(n)$ *e o comprimento esperado do caminho de busca de qualquer elemnento particular é no máximo* $2\log n + O(1)$.

4.5 Discussão e Exercícios

Skiplists foram introduzidas por Pugh [60] que também apresentou numerosas aplicações e extensões da skiplists [59]. Desde então elas têm sido estudadas extensivamente. Diversos pesquisadores têm feito precisas análises do comprimento esperado e da variância do caminho de busca para o i-ésimo elemento em uma skiplist [45, 44, 56]. Versões determinísticas [53], tendenciosas [8, 26], e de ajuste próprio [12] das skiplists têm sido desenvolvidas. Implementações da Skiplist têm sido escritas em diversas linguagens e frameworks e têm sido usadas em sistemas de banco de dados open-source [69, 61]. Uma variante da skiplist é usada na estrutura de gerência de processos do núcleo do sistema operacional HP-UX [42].

Exercício 4.1. Explique os caminhos de pesquisa para 2.5 e 5.5 na skiplist da Figura 4.1.

Exercício 4.2. Explique a adição dos valores 0,5 (com uma altura de 1) e 3,5 (com uma altura de 2) na skiplist da Figura 4.1.

Exercício 4.3. Explique a remoção dos valores 1 e 3 na skiplist da Figura 4.1.

Exercício 4.4. Explique a execução de remove(2) na SkiplistList da Figura 4.5.

Exercício 4.5. Ilustre a execução de add$(3,x)$ na SkiplistList da Figura 4.5, assumindo que o método pick_height() seleciona uma altura de 4 para o nó recém-criado.

Exercício 4.6. Mostre que, durante uma operação add(x) ou remove(x), o número esperado de ponteiros em SkiplistSet que são alterados é constante.

Exercício 4.7. Suponha que, em vez de promover um elemento de L_{i-1} para L_i com base num lançamento de moeda, promovamos com alguma probabilidade p, $0 < p < 1$.

1. Mostre que, com esta modificação, o comprimento esperado de um caminho de pesquisa é no máximo $(1/p)\log_{1/p} n + O(1)$.

2. Qual é o valor de p que minimiza a expressão anterior?

3. Qual é a altura esperada da skiplist?

4. Qual é o número esperado de nós na skiplist?

Exercício 4.8. O método find(x) em uma SkiplistSet às vezes executa *comparações redundantes*; estas ocorrem quando x é comparado com o mesmo valor mais de uma vez. Elas podem ocorrer quando, para algum nó, u, $u.next[r] = u.next[r-1]$. Mostre como essas comparações redundantes acontecem e modifique find(x) para que elas sejam evitadas. Analise o número esperado de comparações feitas pelo seu método find(x) modificado.

Exercício 4.9. Projete e implemente uma versão de uma skiplist que implemente a interface SSet, mas também permite acesso rápido a elementos por classificação. Ou seja, ele também suporta a função get(i), que retorna o elemento cuja classificação é i no tempo esperado $O(\log n)$. (A classificação de um elemento x em um SSet é o número de elementos no SSet que são menores que x.)

Exercício 4.10. Um *indicador* em uma skiplist é um array que armazena a sequência de nós em um caminho de busca no qual o caminho de busca evolui. (A variável *pilha* no código de add(x) na página 87 é um indicador, os nós sombreados na Figura 4.3 mostram o conteúdo do indicador). Pode-se pensar em um dedo apontando o caminho para um nó na lista mais baixa, L_0.

Uma *busca por indicador* implementa a operação find(x) usando um indicador que percorre a lista até alcançar um nó u, tal que $u.x < x$ e $u.next = nil$ ou $u.next.x > x$, e em seguida realiza uma pesquisa normal para x a partir de u. É possível provar que o número esperado de passos necessários para uma pesquisa por indicador é $O(1+\log r)$, onde r é o número de valores em L_0 entre x e o valor apontado pelo indicador.

Implementar uma subclasse de Skiplist chamada SkiplistWithFinger que implementa operações find(x) usando um indicador interno. Esta subclasse armazena um indicador, que é então usado para que cada operação find(x) seja implementada como uma pesquisa de indicador. Durante cada operação find(x), o indicador é atualizado para que cada operação find(x) use, como ponto de partida, um indicador que aponte para o resultado da operação find(x) anterior.

Exercício 4.11. Escreva um método, truncate(i), que trunca uma SkiplistList na posição i. Após a execução deste método, o tamanho da lista é i e contém apenas os elementos nos índices $0,\ldots,i-1$. O valor de retorno é outra SkiplistList que contém os elementos nos índices $i,\ldots,n-1$. Esse método deve ser executado em um tempo $O(\log n)$.

Exercício 4.12. Escreva um método SkiplistList, absorb(l_2), que toma como argumento uma SkiplistList, l_2, esvazia-a e anexa seu conteúdo, em ordem, ao receptor. Por exemplo, se l_1 contiver a,b,c e l_2 contém d,e,f, depois de chamar l_1.absorb(l_2), l_1 conterá a,b,c,d,e,f e l_2 estará vazia. Esse método deve ser executado em um tempo $O(\log n)$.

Exercício 4.13. Usando as idéias da lista eficiente em termos de espaço, SEList, projete e implemente um SSet eficiente em espaço, SESSet. Para fazer isso, armazene os dados, em ordem, em uma SEList, e armazene os blocos desta SEList em um SSet. Se a implementação SSet original usa $O(n)$ espaço para armazenar n elementos, então SESSet usará espaço suficiente para n elementos mais $O(n/b+b)$ espaço perdido.

Exercício 4.14. Usando SSet como sua estrutura subjacente, projete e implemente um aplicativo que leia um arquivo de texto (grande) e permita pesquisar, de forma interativa, qualquer subcadeia contida no texto. À medida que o usuário digita sua consulta, uma parte correspondente do texto (se houver) deve aparecer como resultado.
Dica 1: Cada substring é um prefixo de algum sufixo, então basta armazenar todos os sufixos do arquivo texto.
Dica 2: Qualquer sufixo pode ser representado de forma compacta como um inteiro simples indicando onde o sufixo começa no texto.

Teste sua aplicação em alguns textos grandes, como alguns dos livros disponíveis no Project Gutenberg [1]. Se for feito corretamente, suas aplicações serão bem responsivas; não deve haver atraso notável entre as teclas de digitação e os resultados.

Exercício 4.15. (Este exercício deve ser feito depois de ler sobre árvores de busca binária, em Seção 6.2.) Compare as skiplists com árvores de pesquisa binária das seguintes maneiras:

1. Explicar como a remoção de algumas arestas de uma skiplists leva a uma estrutura que se parece a uma árvore binária e é semelhante a uma árvore de pesquisa binária.

2. Skiplists e árvores de pesquisa binária usam cada uma o mesmo número de ponteiros (2 por nó). As skiplists fazem um melhor uso desses ponteiros. Explique o porquê.