DCC-IME-USP Tópicos de Análise de Algoritmos
Professor Fernando Mário de Oliveira Filho

Skip Lists

Gláucio Alves de Oliveira
Thales A. B. Paiva

{glaucioaorj, thalespaiva}@gmail.com

25 de junho de 2016

Resumimos os principais resultados sobre Skip Lists aleatorizadas (e em breve determinísticas). Skip Lists foram introduzidas por W. Pugh como alternativa às árvores balanceadas. Seu uso é justificado pelos algoritmos mais eficientes e de fácil implementação.

Sumário

1 Introdução **2**

2 *Skip Lists* Probabilísticas **4**
2.1 Definição 4
2.2 Estruturas 5
2.2.1 Nós 5
2.2.2 *Skip Lists* 5
2.3 Algoritmos 5
2.3.1 Inicialização 6
2.3.2 Busca 6
2.3.3 Inserção 7
2.3.4 Remoção 9
2.4 Análise dos Algoritmos 9
2.4.1 Inicialização 9
2.4.2 Busca 10
2.4.3 Inserção 13
2.4.4 Remoção 15

3 *Skip Lists* Determinísticas **15**
3.1 Definições 15
3.2 Estruturas 17
3.2.1 Células 17
3.2.2 SkipList 17
3.3 Algoritmos 18
3.3.1 Inicialização 18
3.3.2 Busca 18
3.3.3 Inserção 19
3.3.4 Remoção 20
3.4 Análise dos algoritmos 22
3.4.1 Inicialização 22
3.4.2 Busca 23
3.4.3 Inserção 23
3.4.4 Remoção 23

1 Introdução

Introduzidas por Pugh [Pugh, 1990], *skip lists* são uma extensão natural de listas ligadas ordenadas. Ambas podem ser usadas para manter um conjunto ordenado de chaves, e eventualmente seus valores associados. Mostramos nas próximas seções que as operações de busca, inserção, e remoção em *skip lists* são competitivas contra as respectivas para árvores balanceadas de busca. Isso faz com que a escolha entre usar *skip lists* ou árvores balanceadas numa determinada aplicação seja baseada nas dificuldades de implementação, no uso de memória, e no tamanho das constantes multiplicativas dos custos de operação.

Uma *skip list* consiste em um conjunto de nós, que têm chave e dados associados, e um conjunto de apontadores com origem em nós de chave menor, e destino em nós de chave maior. Enquanto nas listas ligadas, um nó aponta a um só outro nó, em *skip lists*, um nó pode apontar a vários outros nós de chave maior.

A interface de operações básicas permitidas sobre uma lista ligada é a mesma que a interface para árvores de busca:

- INSERT(S, k, d): Insere um nó de chave k, e conjunto de dados d, em S.
- REMOVE(S, k): Remove o nó de chave k de S.
- SEARCH(S, k): Busca o nó de chave k em S e o devolve.

A Figura 1.1 mostra um exemplo de *skip list*. O nó de chave $-\infty$ indica o começo da lista, e o nó de chave $+\infty$ indica o final. Note também que cada nível de 0 a 3, representado por cada $S_0, \ldots, S_3$, forma uma lista ligada ordenada.

Os dois tipos de *skip lists* são probabilísticas e determinísticas. Primeiro Pugh apresentou as probabilísticas [Pugh, 1990] como alternativa probabilísticas alternativas a árvores balanceadas. Embora de implementação simples, e de ter custo médio de operações

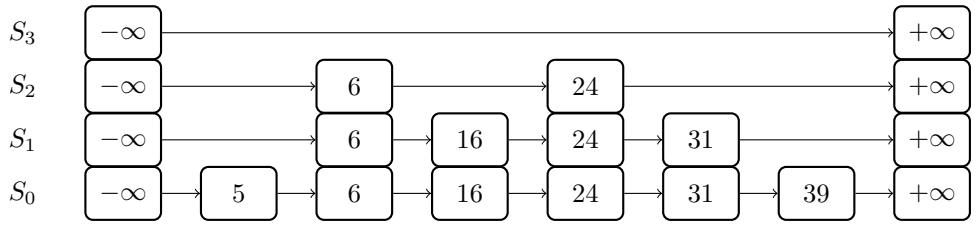

Figura 1.1: Exemplo de skip list.

logarítmico, as operações em *skip lists* probabilísticas têm alto custo no pior caso. Assim, Munro, Papadakis e Sedgewick propõe *skip lists* determinísticas [Munro et al., 1992] que, através de algoritmos de balanceamento, mantêm o custo logarítmico das operações, mesmo no pior caso.

Nas próximas seções, apresentamos as *skip lists* em ordem cronológica.

Estamos prontos para dar uma definição mais precisa de *skip lists*. A definição apresentada não é diretamente implementável em computadores, pois usa conjuntos infinitos. Foi parcialmente baseada na dissertação de mestrado de Mendes [das Chagas Mendes, 2008].

Definição 1.1. *Uma* **Skip List** *de elementos inteiros $c_1, c_2, \ldots, c_n$, em ordem crescente é um conjunto S tal que:*

- $S = \{S_0, S_1, S_2, \ldots\}$ *são os níveis da skip list.*
- $S_0 = \langle -\infty, c_1, c_2, \ldots, c_n, \infty \rangle$.
- $\{-\infty, \infty\} \subseteq S_i$ *para todo i.*
- *Cada S_i, para $i \geq 1$, é um subconjunto ordenado do conjunto S_{i-1}, tal que:*

$$c \notin S_i \Rightarrow c_i \notin S_{i+1}.$$

Definição 1.2. *A* ***altura de um nó*** *c de uma skip list S, denotada por $h(c)$ é o número de conjuntos de S a que c pertence.*

No exemplo da Figura 1.1, $h(16) = 2$.

Definição 1.3. *A* ***altura de uma de uma skip list*** *S, denotada por $h(S)$ é a maior altura de seus elementos finitos. Simbolicamente*

$$h(S) = \max\{h(c) : c \in S_0, c \notin \{-\infty, \infty\}\}.$$

No exemplo da Figura 1.1, $h(S) = 3$.

2 *Skip Lists* Probabilísticas

2.1 Definição

Numa *skip list* probabilística, o número máximo de nós a que um nó pode apontar é definido aleatoriamente quando este é inserido na lista, e é chamado de nível do nó. Na definição do nível de um nó, usa-se um algoritmo que garante que, em média, a proporção de nós com ao menos $i+1$ níveis em relação àqueles com ao menos i níveis, seja de p, uma probabilidade fixada na criação da *skip list*.

Na instanciação de uma *skip lists*, é passada uma probabilidade p, usada na criação dos nós dessa *skip list*. Assim, a adicionamos a seguinte operação à interface dessa estrutura de dados:

- INIT(p): Cria uma *skip list* com parâmetro p vazia e a devolve.

A definição formal de *skip list* probabilística é dada a seguir.

Definição 2.1. *Uma* **Skip List *probabilística*** *de elementos inteiros $c_1, c_2, \ldots, c_n$, em ordem crescente é um par (S, p) tal que:*

- *$S = \{S_0, S_1, S_2, \ldots\}$ é uma skip list.*
- *Cada S_i, para $i \geq 1$, é um subconjunto ordenado do conjunto S_{i-1}, construído de forma que:*
  1. $\Pr(c \in S_i | c \in S_{i-1},\ c \notin \{-\infty, \infty\}) = p$.
  2. $\Pr(c \in S_i | c \notin S_{i-1},\ c \notin \{-\infty, \infty\}) = 0$.

A seguir, provamos um lema simples que nos ajuda a demonstrar fatos interessantes sobre os custos das operações em *skip lists* probabilísticas.

Lema 2.1. *Numa skip list (S, p), a probabilidade de um nó ter altura maior ou igual a k é p^{k-1}.*

Demonstração. Seja H_N a variável aleatória que representa a altura de um nó. Temos, da definição de *skip list*:

$$\begin{aligned}
\Pr(H_N \geq k | H_N \geq k-1) &= p \\
\frac{\Pr(H_N \geq k \wedge H_N \geq k-1)}{\Pr(H_N \geq k-1)} &= p \\
\frac{\Pr(H_N \geq k)}{\Pr(H_N \geq k-1)} &= p \\
\Pr(H_N \geq k) &= p \Pr(H_N \geq k-1).
\end{aligned}$$

Como temos o caso base $\Pr(H_N \geq 1) = 1 = p^0$

$$\Pr(H_N \geq k) = p^{k-1}.$$

□

2.2 Estruturas

2.2.1 Nós

Como uma *skip list* é composta por nós, primeiro vamos definir a estrutura `Node`:

```
1 typedef struct node_s {
2     int key;
3     int height;
4     struct node_s **levels;
5 } Node;
```

Os atributos de uma instância `node` são tais que:

- `node.key` : é a chave do nó, que é única para cada nó de uma *skip list*.
- `node.height` : é o número de níveis atribuído ao nó em sua criação.
- `node.levels[i]` : é o nó para que `node` aponta em seu `i`-ésimo nível.

Opcionalmente, podemos ter um atributo `value`, com os valores associados a cada nó. Porém, como esse atributo não muda em nada os algoritmos, não o consideramos.

2.2.2 *Skip Lists*

Agora podemos definir a estrutura `SkipList`:

```
1 typedef struct skiplist_t {
2     double p;
3     int height;
4     Node *header;
5     Node *end;
6 } SkipList;
```

Abaixo, as descrições dos atributos de uma instância `skiplist` que representa a *skip list* (S, p):

- `skiplist.p` : é a probabilidade p.
- `skiplist.height` : é o nível do maior nó, e pode variar a cada inserção.
- `skiplist.header` : é o primeiro nó da *skip list* com chave $-\infty$.
- `skiplist.end` : é o último nó da *skip list* com chave $+\infty$.

2.3 Algoritmos

Nesta seção, descrevemos os algoritmos para as operações de *skip lists*. Optamos por usar a linguagem C no lugar de pseudocódigo pois os algoritmos usam muitos vetores e ponteiros, então um pseudo-código teria a sintaxe muito parecida com C, mas sem a óbvia vantagem de ser compilável. Os algoritmos implementados são exatamente os apresentados pelo Pugh [Pugh, 1990]. Algumas ideias de estruturas e implementação vieram com ajuda de [Goodrich et al., 2014].

Todo o código mostrado nas próximas seções foi testado e está funcional. Uma implementação completa pode ser vista em: `github.com/thalespaiva/topalgo/blob/master/skiplist/skiplist.c`.

2.3.1 Inicialização

A inicialização de um nó é bem simples. Atribuimos os valores de sua chave, sua altura, e alocamos espaço para os ponteiros dos seus níveis. Note que a altura não é gerada aleatoriamente nessa função.

```
1  void node_init(Node *node, int key, int height) {
2      node->key = key;
3      node->height = height;
4
5      node->levels = malloc(height*sizeof(*(node->levels)));
6  }
```

Algoritmo 1: Inicialização de um nó.

A inicialização da *skip list* cria uma lista de altura 1 com dois nós. Um nó cabeça de chave $-\infty$ e um nó final de chave $-\infty$. O nó cabeça tem altura como a máxima permitida mas apenas o primeiro nível é inicializado. O nó final pode ter altura 0 pois é um nó sentinela e não aponta para nenhum outro.

```
1  void skiplist_init(SkipList *skiplist, double p) {
2      skiplist->p = p;
3
4      skiplist->header = malloc(sizeof(*(skiplist->header)));
5      skiplist->end = malloc(sizeof(*(skiplist->end)));
6
7      node_init(skiplist->header, NEG_INF, SKIPLIST_MAX_HEIGHT);
8      node_init(skiplist->end, INF, 0);
9
10     skiplist->height = 1;
11     skiplist->header->levels[0] = skiplist->end;
12 }
```

Algoritmo 2: Inicialização de uma skip list.

Na Seção 2.3.3, descrevemos o algoritmo de inserção que usa uma função para gerar alturas pseudo-aleatórias de nós. Uma possível implementação para esta função é dada abaixo. Seu funcionamento é bem simples.

```
1  int skiplist_get_random_node_height(SkipList *skiplist) {
2      int height;
3
4      height = 1;
5      while (rand()/(1.0 +  RAND_MAX) < skiplist->p)
6          height++;
7
8      if (height > SKIPLIST_MAX_HEIGHT)
9          height = SKIPLIST_MAX_HEIGHT;
10     return height;
11 }
```

Algoritmo 3: Geração de alturas.

2.3.2 Busca

O algoritmo de busca por um elemento genérico de chave k numa *skip list* faz o seguinte:

1. Percorre o nível mais alto possível até encontrar um apontador para um nó de chave maior ou igual a k.

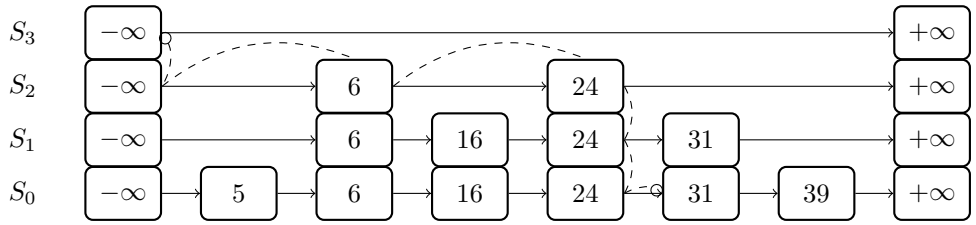

Figura 2.1: Caminho da busca pela chave 31.

2. Quando o próximo nó tem chave maior ou igual a k, desce um nível na busca.
3. Volta para o passo 1 até que o nível de busca seja 0.
4. Quando o nível for 0, estamos olhando para o nó de maior chave, dentre os nós de chave menor ou igual a k. Então, se o próximo elemento tiver chave k, devolva-o. Senão devolva `NULL`.

A Figura 2.1 mostra o caminho da busca pelo elemento 31.

Note que qualquer busca encontra sempre o maior nível do nó de maior chave, dentre os que têm chave menor que a procurada. No nosso exemplo, esse nó é o de chave 24, e a busca encontrou o seu nível 2. Sugerimos que o leitor se familiarize com o algoritmo de busca, pois ele será usado nos algoritmos de inserção e remoção.

O código em C é dado a seguir.

```
1  Node *skiplist_search(SkipList *skiplist, int key) {
2      int i;
3      Node *tmp_node;
4
5      tmp_node = skiplist->header;
6      for (i = skiplist->height - 1; i >= 0; i--) {
7          while (tmp_node->levels[i]->key < key)
8              tmp_node = tmp_node->levels[i];
9      }
10
11     if (tmp_node->levels[0]->key == key)
12         return tmp_node->levels[0];
13
14     return NULL;
15 }
```

Algoritmo 4: Busca.

2.3.3 Inserção

A inserção de um nó k é ligeiramente mais complicada que a busca. Além de buscar a posição da chave k, deve-se fazer com que todos os nós que vêm logo antes em cada nível apontem para o novo nó.

Para isso, enquanto a busca pelo maior nó de chave menor que k é feita, o algoritmo mantém um vetor `pointing_key_node` tal que:

`pointing_key_node[i]` contém o maior nó de chave menor que k no nível `i`.

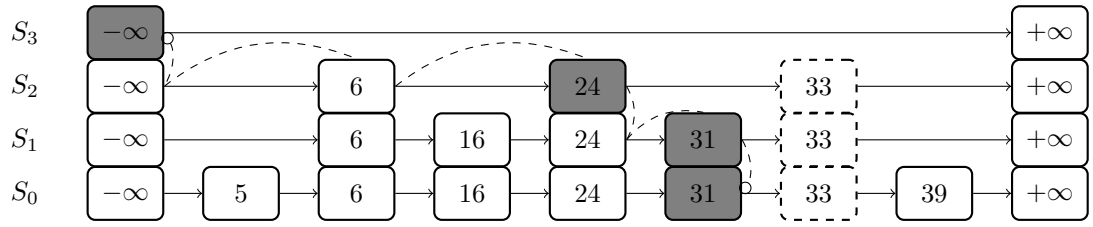

Figura 2.2: Inserção de um nó de chave 33.

Ou seja, `pointing_key_node` contém os nós que, ao final da inserção, apontarão ao novo nó, em cada nível. Note que os nós em `pointing_key_node[i]` não são considerados quando `i` maior ou igual à altura do novo nó.

Uma inserção pode aumentar a altura da *skip list* de h_1 para h_2. Então o algoritmo deve garantir que os níveis entre h_1 e h_2 da cabeça da *skip list* apontem para os respectivos do novo nó.

Como exemplo, considere a Figura 2.2 que trata da inserção de um nó de chave 33. Os elementos de `pointing_key_node` são os retângulos cinza.

Abaixo, o algoritmo em C.

```
1  void skiplist_insert(SkipList *skiplist, int key) {
2      int i;
3      Node *key_node;
4      Node *tmp_node;
5      Node *pointing_key_node[SKIPLIST_MAX_HEIGHT];
6
7      tmp_node = skiplist->header;
8      for (i = skiplist->height - 1; i >= 0; i--) {
9          while (tmp_node->levels[i]->key < key)
10             tmp_node = tmp_node->levels[i];
11         pointing_key_node[i] = tmp_node;
12     }
13
14     tmp_node = tmp_node->levels[0];
15     if (tmp_node->key == key)
16         return;
17
18     key_node = malloc(sizeof(*key_node));
19     node_init(key_node, key, skiplist_get_random_node_height(skiplist));
20
21     if (key_node->height > skiplist->height) {
22         for (i = skiplist->height; i < key_node->height; i++) {
23             pointing_key_node[i] = skiplist->header;
24             skiplist->header->levels[i] = skiplist->end;
25         }
26         skiplist->height = key_node->height;
27     }
28
29     for (i = 0; i < key_node->height; i++) {
30         key_node->levels[i] = pointing_key_node[i]->levels[i];
31         pointing_key_node[i]->levels[i] = key_node;
32     }
33 }
```

Algoritmo 5: Inserção.

2.3.4 Remoção

A remoção é análoga à Inserção. Para remover um nó c, primeiro encontra os nós que apontam a c em cada nível. Depois, faz o nível i de cada um desses nós apontar ao nó que c aponta no nível i. No final, libera a memória ocupada pelo nó removido.

```
1  void skiplist_remove(SkipList *skiplist, int key) {
2      int i;
3      Node *tmp_node;
4      Node *pointing_key_node[SKIPLIST_MAX_HEIGHT];
5
6      tmp_node = skiplist->header;
7      for (i = skiplist->height - 1; i >= 0; i--) {
8          while (tmp_node->levels[i]->key < key)
9              tmp_node = tmp_node->levels[i];
10         pointing_key_node[i] = tmp_node;
11     }
12
13     tmp_node = tmp_node->levels[0];
14     if (tmp_node->key != key)
15         return;
16
17     for (i = 0; i < skiplist->height; i++) {
18         if (pointing_key_node[i]->levels[i] != tmp_node)
19             break;
20         pointing_key_node[i]->levels[i] = tmp_node->levels[i];
21     }
22     free_node(tmp_node);
23 }
```

Algoritmo 6: Remoção.

2.4 Análise dos Algoritmos

Iremos analisar cada um dos algoritmos apresentados. A estrutura da nossa análise é baseada na tese de mestrado de Mendes [das Chagas Mendes, 2008], e também usamos alguns argumentos de Pugh [Pugh, 1990].

2.4.1 Inicialização

Nó

Deve ser claro que a inicialização de um nó é $\mathcal{O}(1)$, pois não tem laços e só faz atribuições simples.

Skip List

Também deve ser claro que a inicialização de uma *Skip List* é $\mathcal{O}(1)$, pois também não tem laços e cada linha é uma atribuição simples ou uma chamada a `node_init`.

Geração de Alturas de Nós

O número esperado de operações feitas pela função `skiplist_get_random_node_height` é proporcional à altura esperada de um nó. Assim, essa chamada tem complexidade $\mathcal{O}(1/p) = \mathcal{O}(1)$.

2.4.2 Busca

Como a função de busca é mais complicada, vamos primeiro definir o custo de uma busca.

Definição 2.2. *O* ***custo de busca*** *de um elemento de uma skip list é o número de comparações feitas durante a busca por ele, sem contar a comparação final, que testa se o nó encontrado tem a chave procurada.*

Assim, o custo de uma busca é o número de vezes que a comparação da linha 7 do algoritmo de busca é executada.

Um conceito que nos ajuda a estudar o custo de buscas é o de caminho de busca. Com ele, podemos analisar o custo de busca separadamente em seus componentes vertical e horizontal.

Definição 2.3. *Um* ***caminho*** *induzido por uma busca é uma sequência* $(o_1, o_2, \ldots, o_m)$ *em que cada passo* o_i *pertence a* $\{\rightarrow, \downarrow\}$*, tal que:*

- *No início da busca, a sequência é vazia.*
- $\rightarrow$ *é adicionado à sequência a cada novo nó visitado na busca.*
- $\downarrow$ *é adicionado à sequência a cada descida de nível na busca.*

Lema 2.2. *O custo de busca de um elemento é igual ao tamanho do caminho de busca por esse elemento.*

Demonstração. Cada $\rightarrow$ é adicionado pela chamada de `tmp_node = tmp_node->levels[i]`, que ocorre quando a condição `tmp_node->levels[i]->key < key` é verificada. Cada $\downarrow$ é adicionado quando ocorre quando a condição `tmp_node->levels[i]->key < key` não é verificada.

Logo, cada comparação adiciona um passo ao caminho de busca, e não há outro modo de adicionar um passo ao caminho de busca. □

Queremos mostrar que o custo médio de busca em *skip lists* é $\mathcal{O}(\log n)$. Pelo lema 2.2, podemos simplificar a demonstração dividindo a busca nas componentes vertical e horizontal.

Lema 2.3. *O número esperado de passos* $\downarrow$ *em qualquer caminho de busca é* $\mathcal{O}(\log n)$*.*

Demonstração. Note que toda busca desce até o nível 0 da *skip list*, então o número esperado de passos é a altura esperada, da *skip list*. Seja H a variável aleatória que representa a altura de uma *skip list* L. E sejam $H_1, H_2, \ldots, H_n$ as variáveis aleatórias que representam as alturas dos elementos finitos $c_1, c_2, \ldots, c_n$, respectivamente.

Como $H = \max\{H_i : i = 1, \ldots, n\}$, é claro que

$$\Pr(H \geq k) \leq \sum_{i=1}^{n} \Pr(H_i \geq k).$$

As H_i seguem a mesma distribuição e essas variáveis são tais que, pelo Lema 2.1, $\Pr(H_i \geq k) = p^{k-1}$. Então

$$\Pr(H \geq k) \leq np^{k-1}.$$

Como a H é discreta e toma valores positivos

$$\begin{aligned}\mathbb{E}(H) &= \sum_{k=0}^{\infty} \Pr(H \geq k) \\ &= \sum_{k=0}^{\lceil 2\log_{1/p} n\rceil - 1} \Pr(H \geq k) + \sum_{k=\lceil 2\log_{1/p} n\rceil}^{\infty} \Pr(H \geq k).\end{aligned}$$

Vamos analisar cada parcela calculada. Para a primeira, temos

$$\begin{aligned}\sum_{k=0}^{\lceil 2\log_{1/p} n\rceil - 1} \Pr(H \geq k) &\leq \sum_{k=0}^{\lceil 2\log_{1/p} n\rceil - 1} 1 \\ &\leq \lceil 2\log_{1/p} n\rceil.\end{aligned}$$

E para a segunda parcela, temos

$$\begin{aligned}\sum_{k=\lceil 2\log_{1/p} n\rceil}^{\infty} \Pr(H \geq k) &\leq \sum_{k=\lceil 2\log_{1/p} n\rceil}^{\infty} np^{k-1} \\ &= n \sum_{k=\lceil 2\log_{1/p} n\rceil}^{\infty} p^{k-1} \\ &= np^{\lceil 2\log_{1/p} n\rceil - 1} \sum_{k=0}^{\infty} p^k \\ &= \frac{n}{p} p^{\lceil 2\log_{1/p} n\rceil} \frac{1}{1-p} \\ &= \frac{n}{p} \left(\frac{1}{p}\right)^{-\lceil 2\log_{1/p} n\rceil} \frac{1}{1-p} \\ &= \frac{n}{p} \left(\frac{1}{p}\right)^{-\lceil 2\log_{1/p} n\rceil} \frac{1}{1-p} \\ &\leq \frac{n}{p} n^{-2} \frac{1}{1-p} \\ &= \frac{1}{np(1-p)}.\end{aligned}$$

Somando os resultados

$$\mathbb{E}(H) \leq \lceil 2\log_{1/p} n\rceil + \frac{1}{np(1-p)}.$$

Portanto

$$\mathbb{E}(H) = \mathcal{O}(\log n).$$

□

Lema 2.4. *O número esperado de passos → em qualquer caminho de busca é* $\mathcal{O}(\log n)$.

Demonstração. Considere que estamos percorrendo ao contrário um caminho de busca por um elemento c qualquer e nos encontramos num nível qualquer de um nó que não c. Temos duas opções:

1. Se possível, subir mais um nível.
2. Se não, ir para a esquerda.

Note que, se for possível subir, o caminho invertido deve subir, caso contrário, não estamos percorrendo um caminho válido.

A probabilidade de ser possível subir mais um nível neste nó é justamente a probabilidade de haver ao menos $i+1$ níveis no nó dado que há ao menos i níveis, ou seja, p. Logo a probabilidade de termos de andar à esquerda é $1-p$.

Seja R_i a variável aleatória que conta o número de passos → feitos no nível i, num caminho de busca. Ou seja, R_i conta o número de observações **antes** de um evento com probabilidade p ser observado. Essa interpretação mostra que $R_i \sim \text{Geom}(p)$.

A esperança de R_i é:

$$\mathbb{E}(R_i) = \sum_{k=1}^{\infty} k \Pr(R_i = k) = \sum_{k=1}^{\infty} k(1-p)^k p$$

Fazendo $q = 1-p$, temos

$$\begin{aligned}
\mathbb{E}(R_i) &= \sum_{k=1}^{\infty} kq^k(1-q) = \sum_{k=1}^{\infty} \left(kq^k - kq^{k+1}\right) \\
&= \sum_{k=0}^{\infty} \left((k+1)q^{k+1} - kq^{k+1}\right) \\
&= \sum_{k=0}^{\infty} q^{k+1} = \sum_{k=1}^{\infty} q^k = \frac{q}{1-q} \\
&= \frac{1-p}{p}.
\end{aligned}$$

E assim, para todo nível i, $\mathbb{E}(R_i) = \mathcal{O}(1)$.

Seja R a variável aleatória que conta o número de passos → no caminho, e seja H a variável aleatória que representa a altura da *skip list*. Então:

$$R = \sum_{i}^{H} R_i.$$

Como propriedade da esperança, temos

$$\mathbb{E}(R) = \mathbb{E}(\mathbb{E}(R|H)).$$

Podemos expandir o termo $\mathbb{E}(R|H)$, obtendo

$$\begin{aligned}
\mathbb{E}(R|H=h) &= \sum_{r=1}^{\infty} r \Pr(R=r|H=h) = \sum_{r=1}^{\infty} r \Pr\left(\sum_{i=1}^{H} R_i = r | H=h\right) \\
&= \sum_{r=1}^{\infty} r \Pr\left(\sum_{i=1}^{h} R_i = r\right) = \sum_{r=1}^{\infty} r \Pr\left(hR_i = r\right) \\
&= \sum_{r=1}^{\infty} r \Pr\left(R_i = \frac{r}{h}\right) = h \sum_{r=1}^{\infty} \frac{r}{h} \Pr\left(R_i = \frac{r}{h}\right) \\
&= h\mathbb{E}(R_i).
\end{aligned}$$

E como $\mathbb{E}(R_i)$ não é uma variável aleatória, temos pela linearidade da esperança que

$$\mathbb{E}(R) = \mathbb{E}(H\mathbb{E}(R_i)) = \mathbb{E}(H)\mathbb{E}(R_i) = \mathcal{O}(\log n)\mathcal{O}(1).$$

Portanto, $\mathbb{E}(R) = \mathcal{O}(\log n)$.

□

Teorema 2.5. *O custo médio de busca em skip lists é* $\mathcal{O}(\log n)$.

Demonstração. É corolário dos lemas 2.3 e 2.4. Num caminho o número de passos é a soma do número esperado de passos $\downarrow$ com o de passos $\rightarrow$. Como ambos são $\mathcal{O}(\log n)$, a soma também é $\mathcal{O}(\log n)$. □

2.4.3 Inserção

Vamos dividir o algoritmo de inserção em três partes:

1. Busca do maior elemento de chave menor do que a chave do nó a ser inserido, com cálculo do vetor `pointing_to_key_node`. Código da linha 7 até a linha 16.
2. Atualização da altura da *skip list*, se necessário. Código da linha 21 até a linha 27.
3. Inserção do nó na *skip list*. Código da linha 29 até a linha 32.

É fácil perceber que o custo da função de inserção é composto pela soma dos custos das partes, mais a soma de um custo constante causado pelas linhas de custo $\mathcal{O}(1)$, que estão fora das partes de interesse.

Parte 1

```
1       tmp_node = skiplist->header;
2       for (i = skiplist->height - 1; i >= 0; i--) {
3           while (tmp_node->levels[i]->key < key)
4               tmp_node = tmp_node->levels[i];
5           pointing_key_node[i] = tmp_node;
6       }
7
8       tmp_node = tmp_node->levels[0];
9       if (tmp_node->key == key)
10          return;
```

Algoritmo 7: Inserção Parte 1.

Seja H novamente a variável aleatória que representa o tamanho de uma *skip list*. O custo da parte 1 é o custo de uma busca mais H vezes o custo de uma atualização de posição de um vetor, que é $\mathcal{O}(1)$.

Assim, seja P_1 o custo da parte 1. Temos que:

$$\mathbb{E}(P_1) = \mathcal{O}(\log n) + \mathbb{E}(H) = \mathcal{O}(\log n) + \mathcal{O}(\log n) = \mathcal{O}(\log n).$$

Parte 2

```
1       if (key_node->height > skiplist->height) {
2           for (i = skiplist->height; i < key_node->height; i++) {
3               pointing_key_node[i] = skiplist->header;
4               skiplist->header->levels[i] = skiplist->end;
5           }
6           skiplist->height = key_node->height;
7       }
```

Algoritmo 8: Inserção Parte 2.

Seja P_2 a variável aleatória que representa o custo da parte 2. P_2 é sempre menor ou igual ao custo no pior caso. Sejam H e H_N as variáveis aleatórias que representam as alturas da *skip list* e do nó a ser adicionado, respectivamente. Temos que

$$P_2 \leq 2(H_N - H) + 1 \leq 2H_N + 1$$

Então podemos afirmar, sobre o custo esperado, que

$$\mathbb{E}(P_2) \leq 2\mathbb{E}(H_N) + 1 = \frac{2}{p} + 1 = \mathcal{O}\left(\frac{2}{p} + 1\right) = \mathcal{O}(1).$$

Parte 3

```
1       for (i = 0; i < key_node->height; i++) {
2           key_node->levels[i] = pointing_key_node[i]->levels[i];
3           pointing_key_node[i]->levels[i] = key_node;
4       }
```

Algoritmo 9: Inserção Parte 3.

Sejam P_3 e H_N as variáveis aleatórias que representam o custo da parte 3, e a altura do nó a ser inserido. Temos que:

$$P_3 = 2H_N$$
$$\mathbb{E}(P_3) = 2\mathbb{E}(H_N) = \frac{2}{p} = \mathcal{O}(1).$$

Juntando os resultados das partes, temos que o custo esperado de uma inserção é:

$$\mathbb{E}(P_1 + P_2 + P_3) = \mathbb{E}(P_1) + \mathbb{E}(P_2) + \mathbb{E}(P_3) = \mathcal{O}(\log n).$$

2.4.4 Remoção

O algoritmo da remoção é muito parecido com o da inserção. A diferença está no trecho de código da linha 17 até a linha 21, copiado abaixo:

```
1       for (i = 0; i < skiplist->height; i++) {
2           if (pointing_key_node[i]->levels[i] != tmp_node)
3               break;
4           pointing_key_node[i]->levels[i] = tmp_node->levels[i];
5       }
6       free_node(tmp_node);
```

Algoritmo 10: Parte exclusiva da Remoção

O número de operações é proporcional à altura da *skip list*, que tem valor esperado $\mathcal{O}(\log n)$. E a função que libera um nó o faz em $\mathcal{O}(1)$, já que é só uma chamada a `free(node->levels)`. Então, se R e I forem as variáveis aleatórias que representam os custos esperados da remoção e da inserção, respectivamente, temos que, para alguma constante c:

$$R \leq I + c(\log n + 1)$$
$$\mathbb{E}(R) \leq \mathbb{E}(I) + c(\log n + 1) = \mathcal{O}(\log n) + \mathcal{O}(\log n) = O(\log n).$$

3 *Skip Lists* Determinísticas

3.1 Definições

Vimos que as operações sobre *skip lists* têm bom custo médio. Porém, apesar de raros, os piores casos de busca, inserção, e remoção têm custos $\mathcal{O}(n)$. Como implementação alternativa de *skip lists* que garantem custo logarítmico até no pior caso, Munro, Papadakis, e Sedgewick, apresentaram essencialmente duas variações de *skip lists* [Munro et al., 1992]:

- 1-2 *Skip List*;
- Topo-Base 1-2-3 *Skip List*, abreviada por TB 1-2-3.

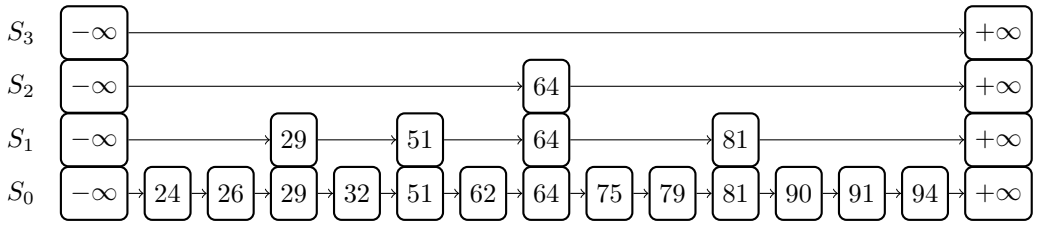

Figura 3.1: Exemplo de TB 1-2-3 *Skip List*

Para cada uma das duas, propuseram implementações das listas de ponteiros horizontais usando listas ligadas e vetores. Ambas usam estruturas de dados parecidas, e, apesar de terem algoritmos simples para a inserção, os algoritmos de remoção são bem complicados, lembrando algoritmos de balanceamento de árvores balanceadas.

Neste texto, tratamos detalhadamente apenas das Topo Base 1-2-3 *Skip Lists*, por serem as mais comumente implementadas, e ter algoritmo de balanceamento mais simples do que as 1-2 *Skip Lists*. O motivo de as TB 1-2-3 serem mais comummente implementadas, é o fato de que o balanceamento pode ser feito de cima para baixo, daí o nome de Topo-Base. Por ter uma implementação mais direta, escrevemos os algoritmos usando listas ligadas para armazenar os ponteiros horizontais de cada nó, pois os algoritmos ficam mais simples.

Uma TB 1-2-3 *Skip List* é uma *skip list* em que, entre quaisquer nós de altura maior ou igual a h consecutivos, há no mínimo 1, e no máximo 3, nós de altura $h-1$. Convencionamos que os níveis dos nós sentinela, $-\infty$ e $+\infty$, são de 1 a mais do que a altura da *skip list*. Um exemplo pode ser visto na Figura 3.1.

A seguir, damos uma definição formal das TB 1-2-3 *skip lists*.

Definição 3.1. *Uma* ***TB 1-2-3*** **Skip List** ***determinística*** *de elementos inteiros $c_1, c_2, \ldots, c_n$, em ordem crescente é um conjunto S tal que:*

- $S = \{S_0, S_1, S_2, \ldots\}$ *é uma skip list.*
- $h(-\infty) = h(+\infty) = h(S) + 1$*, por definição.*
- *Para todo S_i, com $i \geq 1$, se c e d são elementos consecutivos de S_i, então vale que:*

$$1 \leq \Big| \{x \in S_{i-1} : c < x < d\} \Big| \leq 3.$$

Como consequência da definição, temos o seguinte lema sobre a altura de uma 1-2 *skip list*.

Lema 3.1. *A altura de uma TD 1-2-3 skip list é $\mathcal{O}(\log n)$.*

Demonstração. É fácil ver que altura máxima é atingida quando, para cada dois elementos consecutivos c e d, do nível S_i ($i \geq 1$), existir apenas 1 elemento x de S_{i-1} tal que $c < x < d$. Assim, para $i \geq 1$, temos que:

$$|S_i| \leq \frac{|S_{i-1}|}{2} \quad \stackrel{|S_0|=n}{\Longrightarrow} \quad |S_i| \leq \frac{n}{2^i}.$$

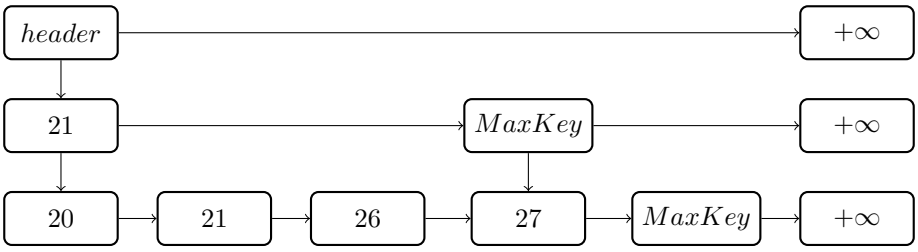

Figura 3.2: Exemplo do uso da estrutura `cell`

Se h for a altura da *skip list*, então $1 \leq |S_{k-1}| \leq 2$. Assim, temos que:

$$1 \leq \frac{n}{2^h} \Rightarrow 2^h \leq n \Rightarrow h \leq \log_2(n) \Rightarrow h = \mathcal{O}(\log n).$$

□

3.2 Estruturas

Os dois tipos de *skip lists* determinísticas usam as mesmas estruturas, definidas nas curtas seções a seguir. Note o contraste com as *skip lists* probabilísticas. Enquanto nelas temos a estrutura nó que representa cada item de uma *skip list*, nas determinísticas temos a estrutura célula, que representa um determinado nível de um dado nó.

3.2.1 Células

```
1  typedef struct cell_t {
2      int key;
3      struct cell_t *down;
4      struct cell_t *right;
5  } Cell;
```

Algoritmo 11: Estrutura das Células

Apesar de parecer uma estrutura familiar, comparando com as *skip lists* probabilísticas, há uma diferença semântica fundamental. O ponteiro `down` da célula `cell` do i-ésimo nível de um nó, aponta para a primeira célula no nível $i-1$ que é pulada por um ponteiro até `cell`.

A Figura 3.2 mostra a representação de uma *skip list* usando esses ponteiros.

3.2.2 SkipList

```
1  typedef struct skiplist_t {
2      Cell *header;
3      Cell *end;
4      Cell *bottom;
5  } SkipList;
```

Algoritmo 12: Estrutura das Skip Lists determinísticas

3.3 Algoritmos

Os algoritmos mostrados nesta seção são baseados no trabalho de Munro et al. sobre *skip lists* determinísticas [Munro et al., 1992], e também na tese de doutorado de Papadakis [Papadakis, 1993]. Neste artigo, os autores descrevem em alto nível os algoritmos de inserção e remoção. Pode-se ver implementações completas em

3.3.1 Inicialização

A inicialização de células é bem simples. Passa-se um ponteiro de célula, uma chave, e ponteiros para as células de baixo e à direita.

```
1
2  void cell_init(Cell *cell, int key, Cell *bottom, Cell *right) {
3      cell->key = key;
4      cell->bottom = bottom;
5      cell->right = right;
6  }
```

Algoritmo 13: Inicialização de células de Skip Lists determinísticas

A inicialização de *skip lists* é simples, mas deve-se atentar para os nós sentinelas, e os valores de seus atributos.

```
1  #define INF INT_MAX
2
3  void skiplist_init(SkipList *skiplist) {
4      skiplist->header = malloc(sizeof(*(skiplist->header)));
5      skiplist->end = malloc(sizeof(*(skiplist->end)));
6      skiplist->bottom = malloc(sizeof(*(skiplist->bottom)));
7
8      cell_init(skiplist->bottom, INF, skiplist->bottom, skiplist->bottom);
9      cell_init(skiplist->end, INF, NULL, skiplist->end);
10     cell_init(skiplist->header, INF - 1, skiplist->bottom, skiplist->end);
11 }
```

Algoritmo 14: Inicialização de Skip Lists determinísticas

3.3.2 Busca

O algoritmo de busca em *skip lists* determinísticas é essencialmente o mesmo daquele para as probabilísticas, percorrendo a lista através dos ponteiros `right` e `bottom`.

```
1  Cell *skiplist_search(SkipList *skiplist, int key) {
2      Cell *cell;
3
4      cell = skiplist->header;
5
6      skiplist->bottom->key = key;
7      while (cell->key != key) {
8          if (key < cell->key)
9              cell = cell->down;
10         else
11             cell = cell->right;
12     }
13     if (cell == skiplist->bottom)
14         return NULL;
15     return cell;
```

```
16 }
```

Algoritmo 15: Algortimo de busca para Skip Lists determinísticas

3.3.3 Inserção

A inserção em *skip lists* determinísticas é feita de modo bem elegante. Uma inserção começa-se na cabeça da *skip list*, em seu último nível. Primeiro, faz-se uma busca, naquele nível, pela chave do elemento a ser inserido. Com isso, descobre-se qual o intervalo que conterá o novo item. Se, em seu nível abaixo, esse intervalo contiver 3 elementos seguidos de mesmo nível, o do meio é aumentado em 1 nível. Esse procedimento é feito a cada nível, até atingir a base da *skip list*. Por fim, pode-se garantidamente inserir o elemento com altura 1.

A Figura 3.3 exemplifica a inserção do elemento 31. Os passos são os seguintes:

- Na iteração do nível 3, vê-se que não há três nós de mesma altura no nível 2. Assim, pode-se descer sem problemas.
- Na iteração do nível 2, vê-se que o intervalo em que estará o 31 é o de $-\infty$ a 64. Como este intervalo têm três elementos de mesma altura, no nível 1, aumenta-se o nível de 29.
- Na iteração do nível 1, vê-se que o intervalo em que estará o 31 é o de 29 a 37. Como não há três elementos de mesma altura no nível 0, pode-se descer ao nível 0.
- Na iteração do nível 0, coloca-se o 31 após o 29, com altura 1.

O algoritmo de inserção é dado a seguir. Pode-se ver como os nós sentinela, ao invés de ponteiros nulos, ajudam a eliminar condicionais antes de fazer buscas de ponteiros, deixando o código mais limpo.

```
1  void skiplist_insert(SkipList *skiplist, int key) {
2      int gap_last_key;
3      Cell *cell, *new;
4
5      skiplist->bottom->key = key;
6
7      cell = skiplist->header;
8      while (cell != skiplist->bottom) {
9
10         while (key > cell->key)
11             cell = cell->right;
12
13         if ((cell->down == skiplist->bottom) && (cell->key == key))
14             return;
15
16         gap_last_key = cell->down->right->right->right->key;
17         if (cell->down == skiplist->bottom || cell->key == gap_last_key) {
18             new = malloc(sizeof(*new));
19             cell_init(new, cell->key, cell->down->right->right, cell->right);
20             cell->right = new;
21             cell->key = cell->down->right->key;
22         }
```

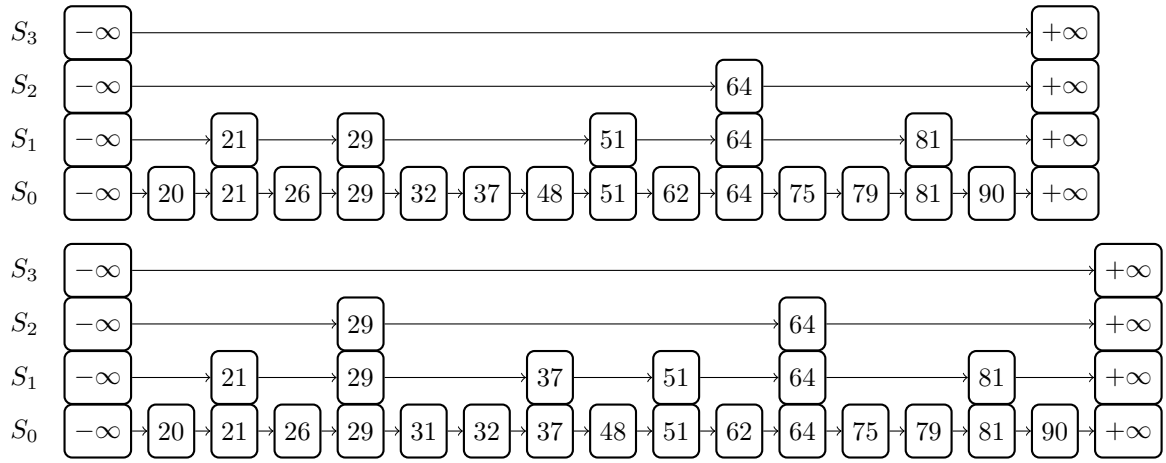

Figura 3.3: Inserção do elemento 31 num TB 1-2-3 skip list.

```
23
24          cell = cell->down;
25      }
26
27      if (skiplist->header->right != skiplist->end)
28          skiplist_grow(skiplist);
29  }
30
31  void skiplist_grow(SkipList *skiplist) {
32      Cell *new_cell;
33
34     new_cell = malloc(sizeof(*new_cell));
35     cell_init(new_cell, INF - 1, skiplist->header, skiplist->end);
36     skiplist->header = new_cell;
37  }
```

Algoritmo 16: Algoritmo de inserção em TB 1-2-3 skip list

3.3.4 Remoção

A remoção em *skip lists* determinísticas é mais complicada, como em árvores balanceadas. A ideia é similar à inserção: a cada nível, observar o nível abaixo fazendo, se necessário, algum balanceamento antes de descer um nível.

A remoção de um elemento começa na cabeça da list, no último nível **da skip list**. Faz-se uma busca nesse nível pelo intervalo em que está o elemento a ser removido. Verifica quantos elementos seguidos de mesma altura há no nível abaixo. Se há 2 ou 3, é seguro descer, pois a remoção será segura. Se há 1, faz-se o balanceamento através da união com um intervalo vizinho, ou troca de altura com um elemento de intervalo vizinho.

Considere a remoção representada na Figura 3.4.

- Na iteração do nível 2, vê-se que 71 está entre $-\infty$ e 75. Também observa-se que,

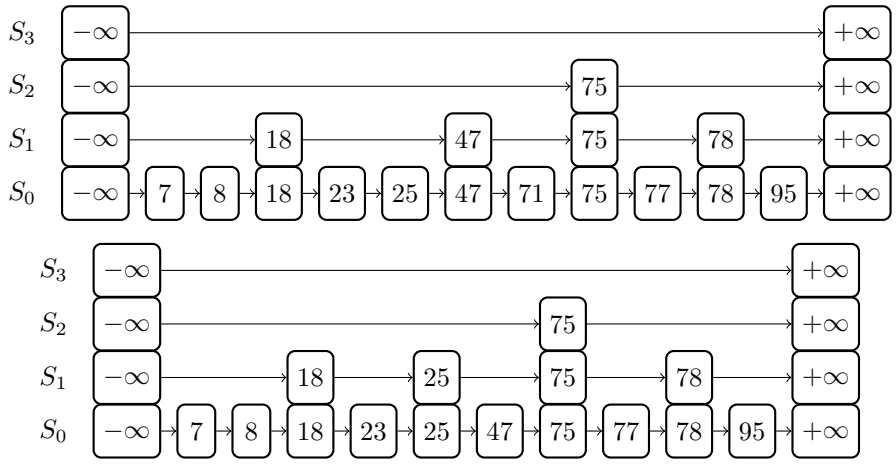

Figura 3.4: Remoção do elemento 71

no nível 1, há dois elementos de mesma altura seguidos, nesse intervalo. Então a descida é segura.

- Na iteração do nível 1, vê-se que 71 está entre 47 e 75. Nesse intervalo, há somente 1 elemento no nível 0. Assim, é preciso balancear para que a remoção seja segura. Como esse intervalo tem um intervalo vizinho com dois elementos, toma-se o elemento mais à direita, 25, para trocar de nível com o 47. Assim, o novo intervalo, de 25 a 75 é seguro, e pode-se descer.

- Na iteração do nível 0, o elemento 71 pode ser removido com segurança.

A seguir, é dado o algoritmo para a remoção. Ele é consideravelmente mais extenso que o de inserção, porém, não é muito mais complicado.

```
1  void skiplist_remove(SkipList *skiplist, int key) {
2      Cell *current, *previous, *next, *node;
3      int previous_key, last_above;
4
5      current = skiplist->header->down;
6      skiplist->bottom->key = key;
7      last_above = skiplist->header->key;
8
9      while (current != skiplist->bottom) {
10         while (key > current->key) {
11             previous = current;
12             current = current->right;
13         }
14         next = current->down;
15
16         if (current->key == next->right->key) {
17             if (current->key != last_above) {
18                 node = current->right;
19                 if ((node->key==node->down->right->key) || (next==skiplist->bottom
                       )) {
20                     current->right = node->right;
```

```
21                      current->key = node->key;
22                      free(node);
23                  }
24                  else {
25                      current->key = node->down->key;
26                      node->down = node->down->right;
27                  }
28              }
29              else {
30                  if (previous->key <= previous->down->right->key) {
31                      if (next == skiplist->bottom)
32                          previous_key = previous->key;
33                      previous->right = current->right;
34                      previous->key = current->key;
35                      free(current);
36                      current = previous;
37                  }
38                  else {
39                      if (previous->key == previous->down->right->right->key)
40                          node = previous->down->right;
41                      else
42                          node = previous->down->right->right;
43                      previous->key = node->key;
44                      current->down = node->right;
45                  }
46              }
47          }
48          last_above = current->key;
49          current = next;
50      }
51      current = skiplist->header->down;
52      while (current != skiplist->bottom) {
53          while (key > current->key)
54              current = current->right;
55          if (key == current->key)
56              current->key = previous_key;
57          current = current->down;
58      }
59
60      if (skiplist->header->down->right == skiplist->end) {
61          current = skiplist->header;
62          skiplist->header = current->down;
63          free(current);
64      }
65  }
```

Algoritmo 17: Algoritmo de remoção em TB 1-2-3 skip list

3.4 Análise dos algoritmos

Apesar de algoritmos mais complicados, a análise dos algoritmos mais simples para *skip list* determinísticas do que para as probabilísticas. Isso pois os laços principais dos algoritmos são feitos percorrendo a altura, que é $\mathcal{O}(\log n)$, e numa busca, não se dá mais do que três passos à direita.

3.4.1 Inicialização

Deve ser claro que a inicialização é $\mathcal{O}(1)$.

3.4.2 Busca

Sabemos que a altura de uma TB 1-2-3 *skip list* é $\mathcal{O}(\log n)$. Então, no pior caso, o laço principal faz $3O(\log n)$ operações, pois, no máximo, são dados três passos à direita.

3.4.3 Inserção

O custo da inserção é dominado pelo custo da busca. Isso pois, dentro do laço principal, é fácil ver que cada condicional tem custo $\mathcal{O}(1)$, e, pelo mesmo argumento usado na análise da busca, o while interno faz 3 comparações no pior caso. Somando isso com o custo $\mathcal{O}(1)$ da função `skiplist_grow`, temos que o custo total da inserção é $\mathcal{O}(\log n)$.

3.4.4 Remoção

Como na inserção, podemos desconsiderar os custos dos condicionais no laço principal da remoção pois não há laço ou chamada de procedimento, que não o `free`, que tem custo $\mathcal{O}(1)$. Assim, o laço principal da remoção é dominado pelo custo de busca.

É claro que o `while` da linha 52 tem custo igual ao de uma busca, a menos de uma constante multiplicativa. E o condicional da linha 60 é $\mathcal{O}(1)$.

Assim, o custo de remoção é $\mathcal{O}(\log n) + \mathcal{O}(\log n) + \mathcal{O}(1) = \mathcal{O}(\log n)$.

Referências

[das Chagas Mendes, 2008] das Chagas Mendes, H. (2008). *Estruturas de dados concorrentes: um estudo de caso em skip graphs*. PhD thesis, Universidade de Sao Paulo.

[Goodrich et al., 2014] Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2014). *Data Structures and Algorithms in Java*. Wiley Publishing.

[Munro et al., 1992] Munro, J. I., Papadakis, T., and Sedgewick, R. (1992). Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 367–375. Society for Industrial and Applied Mathematics.

[Papadakis, 1993] Papadakis, T. (1993). *Skip lists and probabilistic analysis of algorithms*. University of Waterloo Ph. D. Dissertation.

[Pugh, 1990] Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676.