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

27 de abril 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 *Skip Lists* **3**
1.1 Introdução . . . 3
1.2 Definições . . . 4

2 Estruturas **5**
2.1 Nós . . . 5
2.2 *Skip Lists* . . . 5

3 Algoritmos **6**
3.1 Inicialização . . . 6
3.2 Busca . . . 7
3.3 Inserção . . . 7
3.4 Remoção . . . 9

4 Análise dos Algoritmos **9**
4.1 Inicialização . . . 9
4.2 Busca . . . 10
4.3 Inserção . . . 13
4.4 Remoção . . . 15

5 Próxima Entrega **15**

1 *Skip Lists*

1.1 Introdução

Introduzidas por Pugh em [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 operações de busca, inserção, e remoção em *skip lists* são feitas em $\mathcal{O}(\log n)$, como 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* é determinada por um conjunto de nós, que têm chave e dados associados, e apontadores de nós de chave menor a 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. 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 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*.

A interface de operações básicas permitidas sobre uma lista ligada é a mesma que a interface para árvores de busca, com a adição da probabilidade p passada em sua inicialização:

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

Dentre essas operações, a mais importante é a busca. Isso pois, uma vez que o algoritmo de busca é entendido, as outras funções são facilmente descritas.

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.

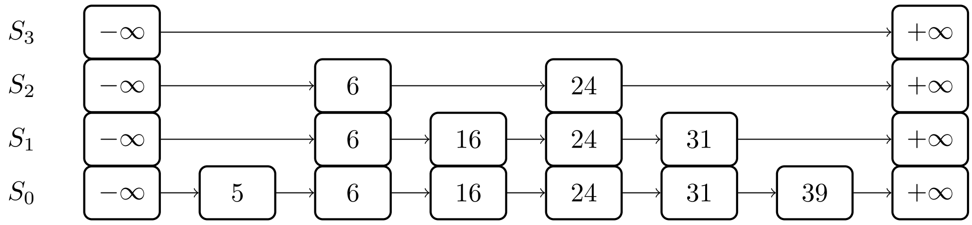

Figura 1.1: Exemplo de skip list.

1.2 Definições

Estamos prontos para dar uma definição mais precisa de *skip lists*, que nos permite provar fatos probabilísticos sobre elas. 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]. Mostramos como implementar a definição abaixo na Seção 2.

Definição 1.1. *Uma* **Skip List** *de elementos inteiros $c_1, c_2, \ldots, c_n$, em ordem crescente é uma dupla (S, p) 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}, construído de forma que:*
  1. $\Pr(c \in S_i | c \in S_{i-1},\ c \notin \{-\infty, \infty\}) = p$.
  2. $\Pr(c \in S_i | c \notin S_{i-1},\ c \notin \{-\infty, \infty\}) = 0$.

Definição 1.2. *Dado um nível S_i de uma skip list, e um elemento c de S_i, o*

Definição 1.3. *A altura de um nó c de uma skip list (S, p), 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 1.4. *A altura de uma de uma skip list $L = (S, p)$, denotada por $h(L)$ é a maior altura de seus elementos finitos. Simbolicamente*

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

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

Lema 1.1. *Numa skip list (S, p), a probabilidade de um nó ter altura maior ou igual a k é p^k.*

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 $\Pr(H_N \geq 1) = p$

$$\Pr(H_N \geq k) = p^k.$$

□

2 Estruturas

2.1 Nós

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

```
typedef struct node_s {
    int key;
    int height;
    struct node_s **levels;
} 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 *Skip Lists*

Agora podemos definir a estrutura `SkipList`:

```
typedef struct skiplist_t {
    double p;
    int height;
    Node *header;
    Node *end;
} 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 $+infty$.

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`.

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.

```c
void node_init(Node *node, int key, int height) {
    node->key = key;
    node->height = height;

    node->levels = malloc(height*sizeof(*(node->levels)));
}
```

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.

```c
void skiplist_init(SkipList *skiplist, double p) {
    skiplist->p = p;

    skiplist->header = malloc(sizeof(*(skiplist->header)));
    skiplist->end = malloc(sizeof(*(skiplist->end)));

    node_init(skiplist->header, NEG_INF, SKIPLIST_MAX_HEIGHT);
    node_init(skiplist->end, INF, 0);

    skiplist->height = 1;
    skiplist->header->levels[0] = skiplist->end;
}
```

Algoritmo 2: Inicialização de uma skip list.

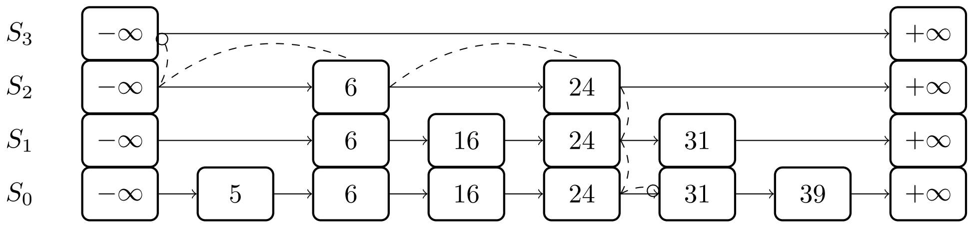

Figura 3.1: Caminho da busca pela chave 31.

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.
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 3.1 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.

```c
Node *skiplist_search(SkipList *skiplist, int key) {
    int i;
    Node *tmp_node;

    tmp_node = skiplist->header;
    for (i = skiplist->height - 1; i >= 0; i--) {
        while (tmp_node->levels[i]->key < key)
            tmp_node = tmp_node->levels[i];
    }

    if (tmp_node->levels[0]->key == key)
        return tmp_node->levels[0];

    return NULL;
}
```

Algoritmo 3: Busca.

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 fazer com que todos os nós que vêm logo antes em cada nível

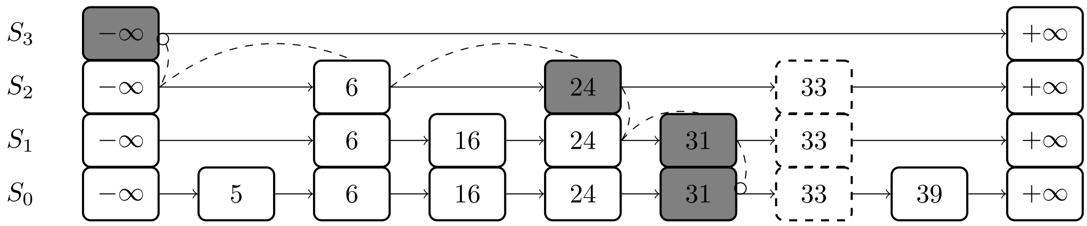

Figura 3.2: Inserção de um nó de chave 33.

apontem para o novo nó.

Para isso, enquanto faz a busca pela pelo maior nó de chave menor que k, 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`.

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 3.2 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.

```c
void skiplist_insert(SkipList *skiplist, int key) {
    int i;
    Node *key_node;
    Node *tmp_node;
    Node *pointing_key_node[SKIPLIST_MAX_HEIGHT];

    tmp_node = skiplist->header;
    for (i = skiplist->height - 1; i >= 0; i--) {
        while (tmp_node->levels[i]->key < key)
            tmp_node = tmp_node->levels[i];
        pointing_key_node[i] = tmp_node;
    }

    tmp_node = tmp_node->levels[0];
    if (tmp_node->key == key)
        return;

    key_node = malloc(sizeof(*key_node));
    node_init(key_node, key, skiplist_get_random_node_height(skiplist));

    if (key_node->height > skiplist->height) {
        for (i = skiplist->height; i < key_node->height; i++) {
            pointing_key_node[i] = skiplist->header;
            skiplist->header->levels[i] = skiplist->end;
        }
        skiplist->height = key_node->height;
    }
```

```

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

Algoritmo 4: Inserção.

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.

```
void skiplist_remove(SkipList *skiplist, int key) {
    int i;
    Node *tmp_node;
    Node *pointing_key_node[SKIPLIST_MAX_HEIGHT];

    tmp_node = skiplist->header;
    for (i = skiplist->height - 1; i >= 0; i--) {
        while (tmp_node->levels[i]->key < key)
            tmp_node = tmp_node->levels[i];
        pointing_key_node[i] = tmp_node;
    }

    tmp_node = tmp_node->levels[0];
    if (tmp_node->key != key)
        return;

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

Algoritmo 5: Remoção.

4 Análise dos Algoritmos

Iremos analisar cada um dos algoritmos apresentados. A estrutura da nossa análise é baseada em [das Chagas Mendes, 2008], mas também usamos alguns argumentos do Pugh [Pugh, 1990].

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`.

4.2 Busca

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

Definição 4.1. *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 4.2. *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 4.1. *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. $\square$

Queremos mostrar que o custo médio de busca em *skip lists* é $\mathcal{O}(\log n)$. Pelo lema 4.1, podemos simplificar a demonstração dividindo a busca nas componentes vertical e horizontal.

Lema 4.2. *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 1.1, $\Pr(H_i \geq k) = p^k$. Então

$$\Pr(H \geq k) \leq np^k.$$

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

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

Somando os resultados

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

Portanto

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

□

Lema 4.3. *O número esperado de passos* $\rightarrow$ *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 $\rightarrow$ 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 $\rightarrow$ no caminho, e seja H a variável aleatória que representa a altura da *skip list*. Então:

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

E temos que

$$\mathbb{E}(R) = \mathbb{E}\left(\sum_{i=1}^{H} R_i\right) = \sum_{i=1}^{\mathbb{E}(H)} \mathbb{E}(R_i) = \mathbb{E}(H)\mathbb{E}(R_i) = \mathcal{O}(\log n)\mathcal{O}(1).$$

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

□

Teorema 4.4. *O custo médio de busca em skip lists é* $\mathcal{O}(\log n)$*.*

Demonstração. É corolário dos lemas 4.2 e 4.3. Como num caminho o número de passos num caminho 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 $\mathcal{O}(\log n)$. □

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 6: 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

```
if (key_node->height > skiplist->height) {
    for (i = skiplist->height; i < key_node->height; i++) {
        pointing_key_node[i] = skiplist->header;
        skiplist->header->levels[i] = skiplist->end;
    }
    skiplist->height = key_node->height;
}
```

Algoritmo 7: 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

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

Algoritmo 8: 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:

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

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:

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

Algoritmo 9: 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:

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

5 Próxima Entrega

Para a próxima entrega, vamos incluir a análise de *skip lists* determinísticas. Depois da análise teórica, queremos comparar empiricamente o custo das *skip lists* com o de árvores balanceadas. Essa comparação será importante pois, como as todas as estruturas têm complexidade logarítmica, uma alteração na constante faz bastante diferença. Uma análise interessante que também queremos colocar é sobre a proporção de cache misses, ou erros na previsão de branch, para cada estrutura.

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.

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