Estruturas de dados concorrentes: um estudo de caso em *skip graphs*

Hammurabi das Chagas Mendes

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação

Orientador: Profa. Dra. Cristina Gomes Fernandes

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, setembro de 2008

Estruturas de dados concorrentes: um estudo de caso em *skip graphs*

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Hammurabi das Chagas Mendes
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Profa. Dra. Cristina Gomes Fernandes – IME-USP
- Prof. Dr. Paulo Feofiloff – IME-USP
- Prof. Dr. Ricardo de Oliveira Anido – UNICAMP

Agradecimentos

Agradeço primeiramente a Deus, que me proporcionou experiências maravilhosas no decorrer do mestrado em São Paulo. Agradeço aos professores, particularmente à professora Cristina Gomes Fernandes, por toda experiência e conhecimento adquiridos. Agradeço, com carinho especial, aos amigos: Diana, Jimena, Johana, Adriana, Wiliam, Andrés, Ivan e Carlos. Vocês são todos inesquecíveis. Muito obrigado à minha noiva, Marcela, por ter acolhido meu amor e por retribuir com mais amor.

Finalmente, aos meus pais, Francisco e Berenice: tudo o que faço é reflexo da linda criação que vocês me deram.

Resumo

Muitos dos sistemas de computação existentes atualmente são concorrentes, ou seja, neles constam diversas entidades que, ao mesmo tempo, operam sobre um conjunto de recursos compartilhados. Nesse contexto, devemos controlar a concorrência das diversas operações realizadas, ou então a interferência entre elas poderia causar inconsistências nos recursos compartilhados ou nas próprias operações realizadas.

Nesse texto, vamos tratar especificamente de *estruturas de dados concorrentes*, ou seja, estruturas de dados cujas operações associadas – consideramos inserção, remoção e busca – sejam passíveis de execução simultânea por diversas entidades. Tendo em vista o controle da concorrência, vamos adotar uma abordagem baseada no emprego de *locks*, uma primitiva de sincronização muito usual na literatura. Nossa discussão será apresentada em termos de certas estruturas de dados chamadas *skip graphs*, que têm propriedades interessantes para outros contextos, como o contexto de sistemas distribuídos.

Palavras-chave: *skip lists*, *skip graphs*, concorrência, análise de complexidade.

Abstract

Many existing computer systems are concurrent, or, in other words, they are composed of many entities that, at the same time, operate over some set of shared resources. In this context, we must control the concurrency of the operations, otherwise the interference between them could cause inconsistencies in the shared resources or in the operations themselves.

In this text, we specifically discuss *concurrent data structures*, or, in other words, data structures over which the associated operations – we consider insertion, removal and search – could be executed simultaneously by various entities. In order to control the concurrency, we will employ an aproach based on the use of locks, a widely known synchronization primitive in the literature. Our discussion will be presented in terms of data structures called skip graphs, which have interesting properties in other contexts, as the context of distributed systems.

Keywords: skip lists, skip graphs, concurrency, complexity analisys.

Conteúdo

Lista de Figuras ix

Lista de Tabelas xi

1 Introdução 1

2 Skip Lists 5

2.1 Estrutura de Dados . . . 5

2.2 Algoritmos . . . 9

2.3 Análise . . . 15

2.3.1 Infra-estrutura . . . 15

2.3.2 Busca . . . 15

2.3.3 Inserção . . . 18

2.3.4 Remoção . . . 18

3 Skip Graphs 19

3.1 Estrutura de Dados . . . 20

3.2 Algoritmos . . . 22

3.3 Análise . . . 30

3.3.1 Busca . . . 30

3.3.2 Inserção . . . 30

3.3.3 Remoção . . . 31

4 Skip Graphs Concorrentes 33

4.1 Modelo 33

4.2 Sincronização 35

4.3 Algoritmos 44

4.3.1 Infra-estrutura 45

4.3.2 Busca 45

4.3.3 Inserção 47

4.3.4 Promoção de *Locks* 48

4.3.5 Escolha das vizinhas compatíveis 50

4.3.6 Remoção 57

4.4 Análise 59

4.4.1 Busca 62

4.4.2 Inserção 63

4.4.3 Remoção 64

5 Desempenho e Aplicação dos *Skip Graphs* 65

5.1 Desempenho dos *Skip Graphs* Concorrentes 65

5.2 Aplicação em Sistemas Distribuídos 71

5.2.1 Compartilhamento Distribuído de Arquivos 75

6 Considerações Finais 77

Bibliografia 79

Lista de Figuras

2.1 *Um exemplo de lista ligada com formato fixo.* 6

2.2 *Um exemplo de skip list.* 9

2.3 *Busca em uma skip list.* 11

2.4 *Busca em uma skip list antes da inserção (preenchendo ante).* 12

2.5 *Inclusão da célula 3 na lista ligada da figura 2.4.* 13

2.6 *Busca em uma skip list antes da remoção (preenchendo ante).* 14

2.7 *Remoção da célula 3 da lista ligada da figura 2.6.* 15

3.1 *Um exemplo de skip graph.* 21

3.2 *Compartilhamento de níveis em um skip graph.* 23

3.3 *Criação de um novo nível no skip graph.* 25

3.4 *Remoção de um nível no skip graph.* 25

3.5 *Busca em um skip graph.* 27

3.6 *Inserção da célula 4 em um nível $i+1$ após a inserção no nível i.* 29

4.1 *Busca por um vizinho compatível em uma única direção.* 52

4.2 *Inserção no nível superior por meio de um único vizinho compatível.* 52

4.3 *Problemas do algoritmo tradicional em ambientes concorrentes.* 53

4.4 *Inserções concorrentes em uma mesma lista ligada.* 56

5.1 *Estado dos processos com $\gamma = 10\%$, $I = 250$, $R = 200$ e $|U| = 350$.* 68

5.2 *Estado dos processos com $\gamma = 50\%$, $I = 250$, $R = 200$ e $|U| = 350$.* 69

5.3 *Estado dos processos com* $\gamma = 100\%$, $I = 250$, $R = 200$ *e* $|U| = 350$. 70

5.4 *Estado dos processos com* $\gamma = 10\%$, $I = 65$, $R = 50$ *e* $|U| = 75$. 71

5.5 *Estado dos processos com* $\gamma = 50\%$, $I = 65$, $R = 50$ *e* $|U| = 75$. 72

5.6 *Estado dos processos com* $\gamma = 100\%$, $I = 65$, $R = 50$ *e* $|U| = 75$. 73

Lista de Tabelas

5.1 *Dados obtidos nas operações, com* $\gamma = 10\%$, $I = 250$, $R = 200$ *e* $|U| = 350$. 68

5.2 *Dados obtidos nas operações, com* $\gamma = 50\%$, $I = 250$, $R = 200$ *e* $|U| = 350$. 69

5.3 *Dados obtidos nas operações, com* $\gamma = 100\%$, $I = 250$, $R = 200$ *e* $|U| = 350$. 70

5.4 *Dados obtidos nas operações, com* $\gamma = 10\%$, $I = 65$, $R = 50$ *e* $|U| = 75$. 71

5.5 *Dados obtidos nas operações, com* $\gamma = 50\%$, $I = 65$, $R = 50$ *e* $|U| = 75$. 72

5.6 *Dados obtidos nas operações, com* $\gamma = 100\%$, $I = 65$, $R = 50$ *e* $|U| = 75$. 73

Capítulo 1

Introdução

Muitos dos sistemas de computação existentes atualmente são *multithreaded* ou *distribuídos*, nos quais temos várias entidades que podem operar ao mesmo tempo sobre um conjunto de recursos compartilhados, acessíveis a todas elas. Essas entidades poderiam ser *threads* concorrentes de um único processo, acessando recursos visíveis ao processo em questão, ou mesmo de vários processos, acessando recursos visíveis no âmbito de uma máquina inteira. Essas entidades poderiam também ser processos presentes em diversas máquinas, operando sobre recursos compartilhados entre elas. Mesmo dentro desses processos distribuídos, poderíamos ter várias *threads* concorrentes. O importante é que várias entidades podem acessar ao mesmo tempo um mesmo recurso compartilhado. Há uma variedade de possíveis recursos compartilhados, como arquivos, dispositivos, um conjunto de dados, uma estrutura de dados, etc. Para simplificar nossa terminologia, vamos chamar as entidades anteriores (*threads* ou processos) apenas pelo nome de *processos*. Portanto, temos vários processos acessando recursos compartilhados.

Com relação ao usufruto de recursos, vamos chamar de *ambiente tradicional* o contexto onde temos um único processo que opera sobre qualquer recurso; de *ambiente concorrente* o contexto onde temos vários processos operando sobre recursos compartilhados; e de *ambiente distribuído* o que além de ser concorrente, tem seus processos e recursos em máquinas diferentes e interligadas. Em um ambiente concorrente, os recursos compartilhados estão armazenados normalmente em uma memória acessível a todos os processos presentes. Já em um ambiente distribuído, os recursos compartilhados podem estar espalhados entre as memórias de diversas máquinas. Os "pedaços" do recurso estariam interligados por meio de *apontadores distribuídos*, que permitiriam referenciar dados contidos em máquinas remotas. No nosso texto, vamos tratar de ambientes concorrentes.

Em um ambiente tradicional, quando temos um conjunto de dados nos quais queremos realizar operações de maneira eficiente, empregamos estruturas de dados para impor uma organização em conjuntos de elementos. Desta forma, podemos conceber algoritmos que usufruam dessa organização

para ganharmos eficiência. Em um ambiente concorrente, uma eficiência ainda maior poderia ser obtida se executarmos várias operações ao mesmo tempo. Como veremos adiante, para isso funcionar precisamos de um controle adicional. Em ambientes distribuídos, vamos supor que as diversas células da estrutura estejam dispostas em diversas máquinas, e que as referências da estrutura sejam apontadores distribuídos, conforme mencionamos anteriormente.

O acesso a um recurso qualquer é feito usando as rotinas cabíveis ao recurso em questão. Por exemplo, as operações possíveis em um arquivo poderiam ser "abrir", "fechar", etc. Já as operações possíveis em uma lista ligada poderiam ser "inserir", "remover", "buscar", etc. O problema de que recursos sejam compartilhados trivialmente, isto é, sem que suas operações sejam modificadas, é que tais operações podem deixar de funcionar em ambientes concorrentes, ao interferir umas sobre as outras. Isso ocorre porque em um ambiente tradicional uma única operação é feita por vez, o que não é verdade em ambientes concorrentes. Por exemplo, consideremos uma lista ligada compartilhada. Se um processo A insere um elemento ao mesmo tempo que um processo B remove um elemento, os apontadores da lista ligada poderiam ficar em um estado inválido. Portanto, as operações precisariam ser modificadas para "controlarem" essa concorrência. A concepção das operações associadas a recursos compartilhados deve levar em conta que as outras operações possíveis podem estar sendo executadas concorrentemente.

Em ambientes distribuídos, além do controle da concorrência das diversas operações, existem ainda outras preocupações, como falhas nas máquinas ou no mecanismo de comunicação envolvido, além da concentração de carga entre as diversas máquinas. Mais precisamente:

1. Se temos ambientes distribuídos muito grandes, é de se esperar que possam falhar máquinas ou interligações entre essas máquinas. Assim, uma parte dos recursos compartilhados, que porventura estivessem armazenados em uma máquina que tivesse falhado ou tivesse ficado inalcançável, estaria "perdida". Se, por exemplo, os diversos processos compartilhassem uma estrutura distribuída, então certos apontadores iriam referenciar pedaços perdidos da estrutura, tornando-a inconsistente.

2. Em um ambiente distribuído, é importante a distribuição de carga entre os processos. Nesse contexto, vamos supor que empregássemos uma árvore balanceada distribuída para fazer buscas. Então, as células da árvore estariam armazenadas nas diversas máquinas envolvidas, e seriam acessíveis por apontadores distribuídos. Como a árvore balanceada utilizada tem raiz, toda busca iria passar pelo processo que armazena tal raiz, que concentraria muita carga do ambiente distribuído e penalizaria severamente a escalabilidade da abordagem.

No nosso texto, vamos tratar apenas do problema da concorrência entre diversos processos, e para isso vamos tomar um ambiente concorrente, sem ser distribuído. Dessa forma, também podemos

ignorar problemas como distribuição de carga ou tolerância à falhas. Nos capítulos 2 e 3, começamos mostrando estruturas tradicionais de dados que permitem inserção, remoção e busca de elementos em um conjunto de uma maneira eficiente. A primeira, a *skip list*, serve como base para definir a segunda, o *skip graph*. Os *skip graphs* não têm raiz, e já caminham na direção de serem usáveis em ambientes distribuídos (apesar de tratarmos somente de ambientes concorrentes, esse fator é importante). Na literatura, existem estudos a respeito do controle da concorrência em *skip lists* [26, 30]. O objetivo desse texto é fazer um estudo parecido com relação aos *skip graphs*. Desta forma, o enfoque desse texto é o controle da concorrência nos *skip graphs*.

Capítulo 2

Skip Lists

Neste capítulo, vamos apresentar uma estrutura de dados denominada *skip list*, que possui a mesma funcionalidade de uma árvore balanceada de busca e fundamenta uma outra estrutura de dados, denominada *skip graph*, de que falaremos no próximo capítulo. Vamos apresentar a estrutura considerando um ambiente tradicional, no qual temos um único processador e um único dispositivo de memória, acessível globalmente.

2.1 Estrutura de Dados

Tomemos um conjunto de n elementos distintos, oriundos de um universo totalmente ordenado U. O emprego de uma árvore balanceada de busca permite realizar eficientemente operações de busca, inserção e remoção nesse conjunto. Para isso, o mecanismo de balanceamento controla a distância da raiz às folhas, e garante que as operações mencionadas consumam tempo $O(\lg n)$ [2, 9].

Visando alcançar eficiência semelhante, podemos imaginar uma *lista ligada ordenada com atalhos*, esses utilizados para tornar as buscas velozes. Inicialmente, vamos tentar fixar estes atalhos com base no posicionamento das células, mas veremos que isso teria impacto nas inserções e remoções. Depois, uma abordagem probabilística nos levará às *skip lists*. Essas estruturas foram definidas por Pugh [29], e são alternativas eficientes para árvores balanceadas de busca [31, 36].

Com relação a estruturas de dados, é muito interessante a existência de múltiplas alternativas com funcionalidades similares. Estruturas de dados complexas são usualmente definidas em termos de outras mais simples, e a existência de múltiplas alternativas permite maiores possibilidades, já que características das mais simples são normalmente refletidas e muitas vezes aproveitadas nas mais complexas. Isso será mostrado na prática no próximo capítulo.

Listas Ligadas com Formato Fixo

Considere uma lista ligada ordenada com cabeça, que contenha elementos do conjunto universo U. Para permitir uma busca eficiente, inserimos entre as células dessa lista ligada apontadores adicionais, os *atalhos*, conforme a seguinte idéia:

- A cada duas células, apontamos a que está duas posições à frente;
- A cada quatro células, apontamos a que está quatro posições à frente;
- ...
- A cada 2^i células, apontamos a que está 2^i posições à frente (para $i \geq 1$).

Veja um exemplo desse cenário na figura 2.1. O resultado dessas modificações é o que denominamos uma *lista ligada ordenada com atalhos*, ou LA. Mais especificamente, já que os atalhos têm uma posição bem definida, vamos dizer que temos uma *lista ligada ordenada com atalhos fixos*, ou LAF.

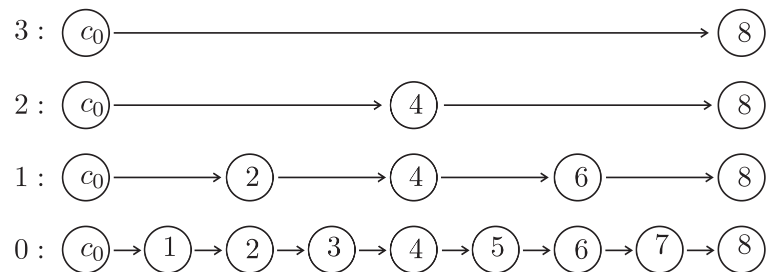

Figura 2.1: *Um exemplo de lista ligada com formato fixo.*

Se um atalho fosse ultrapassar a última célula da lista ligada, vamos estabelecer que ele seja nulo. Vamos criar atalhos com distâncias crescentes enquanto os atalhos que saem da cabeça não ultrapassarem a última célula da LAF, ou seja, enquanto esses atalhos não forem nulos.

Vamos considerar que nossas listas ligadas possuam n elementos. Além disso, vamos considerar que todos os elementos são distintos dois a dois. Em uma LA, é como se houvesse várias listas ligadas ordenadas, de vários níveis diferentes. De baixo para cima, vamos denominar essas listas ligadas por $S_0, \ldots, S_k$. Dizemos que i é o nível da lista ligada S_i, e k, o nível mais alto de uma lista ligada, é o nível da estrutura. Da mesma forma, dizemos que a lista ligada S_i tem ou é de nível i, e que a estrutura tem ou é de nível k. Observe que $k = \lfloor \lg n \rfloor$. A lista ligada S_0 contém todos os elementos, e as listas ligadas S_i são subconjuntos de S_{i-1}, para todo $0 < i \leq k$. Em outras palavras, a lista ligada original de nível 0 é tornada mais esparsa a cada nível da estrutura.

Vamos definir uma célula-elemento como uma célula qualquer com exceção da cabeça. Observe que o número de células-elemento de S_i é a metade (truncada) do número de células-elemento de S_{i-1}, para todo $0 < i \leq k$.

Em uma LA, vamos dizer que uma célula qualquer c tem ou é de nível i se o maior inteiro j tal que c pertence a S_j é i. A chave de uma célula c é denotada por $c.chave$, e o nível de uma célula c é denotado por $c.nivel$. Além disso, cada célula c possui um campo *prox* que é um vetor com $1 + c.nivel$ apontadores para outras células. A cabeça da LA, denotada por c_0, é a única célula que não contém um elemento válido de U. Por motivos técnicos, manteremos $c_0.chave = -\infty$. Em uma célula qualquer c, que pertença a S_i, seu apontador para a célula seguinte em S_i é dito um apontador de nível i e é denotado por $c.prox[i]$. Em uma LAF, podemos perceber que $c_0.nivel = k$, ou seja, a cabeça tem nível máximo, o próprio nível da LAF.

Em uma lista ligada ordenada S (por exemplo, em cada S_i), vamos dizer que uma célula c em S *vem logo antes* de um elemento e em U se $c.chave$ é a maior possível dentre $\{c'.chave : c' \in S \text{ e } c.chave < e\}$. Similarmente, vamos dizer que uma célula c em S *vem logo depois* de um elemento e em U se $c.chave$ é a menor possível dentre $\{c'.chave : c' \in S \text{ e } c.chave > e\}$. Vamos também dizer que uma célula c *vem logo antes* de e incluindo e ou *vem logo depois* de e incluindo e de maneira análoga ao que foi feito antes, mas permitindo que $c'.chave$ seja também igual a e.

Vamos agora calcular a quantidade total T_{LAF} de apontadores entre as $n+1$ células de uma LAF com n elementos:

$$T_{LAF} = \sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{\lfloor \lg n \rfloor} \frac{n}{2^i} = n \sum_{i=0}^{\lfloor \lg n \rfloor} \frac{1}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n = O(n).$$

Já a quantidade A_{LAF} de atalhos (apontadores adicionais, que pertencem às listas ligadas de nível > 0) é de $\Omega(n)$, pois só no nível 1 existem $\lfloor n/2 \rfloor$ desses atalhos. Por isso, $T_{LAF} = \Theta(n)$.

Para buscar um elemento e em uma LAF, partimos da cabeça c_0 e seguimos apontadores de nível k, até a célula que vem logo antes incluindo de e em S_k. Se não encontramos e, descemos para o nível $k-1$, e seguimos apontadores de nível $k-1$, até a célula que vem logo antes de e incluindo e em S_{k-1}. Isso se repete até encontrarmos e ou tentarmos diminuir de nível no nível 0. Neste caso, ao final, estamos na célula que vem logo antes de e incluindo e em S_0. Observando a figura 2.1, é fácil perceber que essa operação em uma LAF é uma busca binária, que sabemos que consome tempo $O(\lg n)$ [9].

O problema, todavia, é que a inserção e a remoção de elementos em uma LAF seriam bastante ineficientes. Por exemplo, ao inserir no início da lista ligada ou remover da primeira posição da lista

ligada, precisaríamos restabelecer todos os apontadores adicionais, refletindo os novos posicionamentos das células. Isto consumiria tempo $\Omega(n)$, pois $A_{LAF} = \Omega(n)$. Precisamos de outra alternativa que não tenha tal característica, mas que ainda proporcione uma busca eficiente.

Listas Ligadas com Formato Aleatório

Em uma LAF, 100% das células-elemento são de nível ≥ 0, 50% delas são de nível ≥ 1, 25% delas são de nível ≥ 2, e assim por diante. A quantidade de células-elemento de nível pelo menos i é sempre a metade da quantidade de células-elemento de nível pelo menos $i - 1$, para cada $0 < i \leq k$. A cabeça sempre tem nível máximo. O arranjo dos atalhos determina os níveis das células na LAF, mas como este arranjo é feito com base no posicionamento dos elementos, ele pode precisar ser refeito globalmente na ocasião de uma inserção ou de uma remoção. Idealmente, qualquer operação deveria ter um escopo de mudança bem localizado. Quando isso acontece bastante em uma estrutura de dados, por mérito de seu *design*, vamos dizer que ela possui *localidade*.

Vamos imaginar uma outra lista ligada ordenada com atalhos, onde os níveis das células-elemento sejam estabelecidos por meio de um procedimento probabilístico, mas seguindo o modelo de distribuição de níveis das LAFs. Em outras palavras, o número esperado de células-elemento no nível i dessa outra lista ligada é o número de células-elemento no nível i de uma LAF de mesmo tamanho, para $0 \leq i \leq k$. Para isso, a probabilidade de uma célula-elemento ser de nível ≥ 0 tem que ser de 100% e a probabilidade de uma célula-elemento ser de nível pelo menos i tem que ser metade da probabilidade dessa célula-elemento ser de nível pelo menos $i - 1$, para $0 < i \leq k$. Na verdade, podemos generalizar o "metade" anterior para um número $0 < p < 1$ qualquer. Essa idéia nos leva à definição das *skip lists*:

Definição 1. *Uma* skip list *com parâmetro p é uma lista ligada ordenada com atalhos onde o nível de cada célula-elemento c é escolhido independente e aleatoriamente, no momento de sua criação, de acordo com a seguinte distribuição de probabilidade:*

$$\Pr[c.nivel = j] \;=\; p^j(1-p), \; para \; j = 0, 1, 2, \ldots$$

A cabeça possui nível máximo, o maior dentre os níveis das células-elemento da skip list*. Para toda célula c da* skip list*, o campo $c.chave$ guarda sua chave, o campo $c.nivel$ guarda seu nível e o campo $c.prox$ é uma tabela com $c.nivel + 1$ entradas, onde $c.prox[i]$ aponta para a célula seguinte a c na lista ligada de nível i.*

Observação 1. *Podemos perceber que:*

1.

$$\Pr[c.nivel \geq j] \;=\; 1-(1-p)\sum_{i=0}^{j-1} p^i = 1-(1-p)\frac{(p^j-1)}{p-1} = 1+p^j-1 = p^j;$$

2.

$$\sum_{j=0}^{\infty} \Pr[c.nivel = j] \;=\; (1-p)\sum_{j=0}^{\infty} p^j = (1-p)/(1-p) = 1.$$

Um exemplo de *skip list* é mostrado na figura 2.2. Essa abordagem utilizada nas LAs visa uma busca tão eficiente quanto nas LAFs, mas com localidade nas inserções e remoções. Nas seções seguintes, vamos descrever e analisar as operações que podemos utilizar nas *skip lists*, e veremos, de fato, que não precisamos de amplos rearranjos de atalhos nas inserções ou remoções.

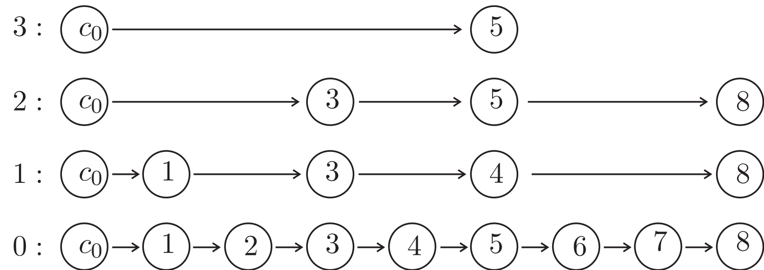

Figura 2.2: *Um exemplo de skip list.*

Por último, podemos perceber que uma *skip list* com parâmetro p pode ser vista, de maneira abstrata, como uma coleção de listas ligadas $\{S_0, \ldots, S_k\}$, onde uma célula-elemento de nível i está presente nas listas ligadas $S_0, \ldots, S_i$ e o nível das células-elemento é definido conforme a definição 1. Esse outro ponto de vista será utilizado no capítulo seguinte.

2.2 Algoritmos

Nessa seção, vamos apresentar os algoritmos de inicialização, busca, inserção e remoção em uma *skip list*. O apontador nulo ao final das listas ligadas é denotado por $\perp$.

Infra-estrutura

O algoritmo 1, `alocarCélula(`e`, `j`)`, permite alocar uma nova célula c para a *skip list*, atribuindo $e \in U$ ao campo $c.chave$ e $j \geq 0$ ao campo $c.nivel$.

Vamos considerar definido um algoritmo `liberarCélula()`, que libera da memória uma célula c, recebida como parâmetro. Consideramos que a célula c foi alocada previamente por `alocarCélula()`.

O algoritmo 2, `escolherNível(`p`)`, usado na inserção de novos elementos na *skip list*, escolhe um

Algoritmo 1 alocarCélula(e, j)

1: $\triangleright$ c é uma nova célula alocada na memória
2: $c.chave \leftarrow e$
3: $c.nivel \leftarrow j$
4: **return** c

nível para novas células com base no parâmetro $0 < p < 1$, segundo a distribuição de probabilidades da definição 1. A função `rand()`, usada neste algoritmo, devolve um número $x \in [0, 1]$, segundo a distribuição uniforme.

Algoritmo 2 escolherNível(p)

1: $j \leftarrow 0$
2: **while** rand() $\leq p$ **do**
3: $\quad j \leftarrow j + 1$
4: **return** j

O algoritmo 3, `inicializar()`, aloca a cabeça da *skip list* e inicializa seus valores, já fazendo com que $k = c_0.nivel$, que sempre vai ocorrer na nossa estrutura.

Algoritmo 3 inicializar()

1: $c_0 \leftarrow$ alocarCélula($-\infty$, 0)
2: $c_0.prox[0] \leftarrow \perp$

Busca

O algoritmo 4, `buscar(`c`, `j`, `e`)`, recebe uma célula c, um nível j e um elemento $e \in U$, de forma que $c.chave \leq e$, e devolve a célula que vem logo antes de e incluindo e na estrutura. O parâmetro j é o nível pelo qual a busca é iniciada. Tipicamente, o algoritmo é invocado com $c = c_0$ e $j = c_0.nivel$.

Este algoritmo mantém sempre o seguinte:

1. c é uma célula com chave $\leq e$;
2. se c for uma célula com nível $= j$, então $c.prox[j + 1] = \perp$ ou $c.prox[j + 1].chave > e$.

O algoritmo 4 foi escrito de maneira recursiva refletindo o caráter recursivo da *skip list*: tomando qualquer célula c e nível j, se as células com chave menor que $c.chave$ e os níveis maiores que j fossem desconsiderados, teríamos uma *skip list* "menor", onde c seria análoga à célula-cabeça e j seria o nível da estrutura. Além disso, o algoritmo permite visualizar claramente que fazemos visitas

laterais enquanto estivermos visitando células que vêm antes do elemento buscado, descendo de nível em caso contrário.

Algoritmo 4 buscar(c, j, e)

1: **if** $c.chave = e$ **then**
2:    **return** c
3: **if** $c.prox[j] \neq \perp$ **and** $c.prox[j].chave \leq e$ **then**
4:    **return** buscar($c.prox[j]$, j, e)
5: **else**
6:    **if** $j \geq 1$ **then**
7:       **return** buscar(c, $j-1$, e)
8:    **else**
9:       **return** c

Um exemplo de busca é mostrado na figura 2.3.

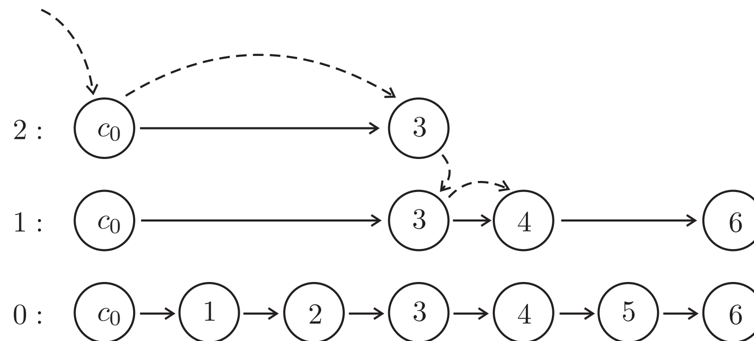

Figura 2.3: *Busca em uma skip list pelo elemento 4.*

Se há uma invocação recursiva do algoritmo `buscar()` com argumento $c.prox[j]$ (algoritmo 4, linha 4), então, pela segunda observação feita anteriormente, $c'.nivel = j$. Logo, sempre que visitamos uma nova célula, caminhando lateralmente em uma lista ligada de um certo nível, estamos visitando células através do maior nível possível. Portanto, podemos descartar o parâmetro j e reescrever o algoritmo 4, `buscar()`, conforme fazemos no algoritmo 5, `buscar2()`.

Inserção

A algoritmo 6, `inserir(`c_0`, `e`)`, recebe a célula-cabeça c_0 de uma *skip list* e um elemento $e \in U$. Se a *skip list* já contém uma célula cuja chave é e, o algoritmo não modifica a estrutura. Caso contrário, o algoritmo insere na posição correta da estrutura uma nova célula cuja chave é e e atualiza todas as referências.

A inserção é feita em duas etapas. Na primeira etapa, entre as linhas 1 e 4 do algoritmo `inserir()`,

Algoritmo 5 buscar2(c, e)

```
1: if c.chave = e then
2:   return c
3: j ← c.nivel
4: while j ≥ 0 do
5:   if c.prox[j] ≠ ⊥ and c.prox[j].chave ≤ e then
6:     return buscar2(c.prox[j], e)
7:   else
8:     j ← j − 1
9: return c
```

fazemos uma busca para verificar a posição do elemento a ser inserido. Nessa busca, armazenamos informações intermediárias que serão úteis na etapa seguinte, que é a inserção em si. Mais especificamente:

(i) Utilizamos a função `alocarTabela()`, que aloca e devolve uma tabela vazia, e a atribuímos à tabela auxiliar *ante*;

(ii) Utilizamos o algoritmo 7, `buscarCriaAnte(`c`, `e`, `*ante*`)`. Ele recebe uma célula c, um elemento $e \in U$ e uma tabela vazia, de forma que $c.chave \leq e$, e devolve a célula que vem logo antes de e incluindo e na estrutura, fazendo também com que $ante[i]$ aponte para a célula que vem logo antes de e na lista ligada S_i, para todo $i \leq c.nivel$. Tipicamente, o algoritmo é invocado com $c = c_0$. O algoritmo é escrito com base na função `buscar2()`, o que irá facilitar sua análise, na seção 2.3. Uma diferença notável entre os dois algoritmos é que agora nos mantemos sempre antes da célula cuja chave é o elemento buscado, e nunca sobre a própria célula, de forma a preencher a tabela passada como argumento de maneira apropriada (compare a linha 5 do algoritmo 5 com a linha 3 do algoritmo 7). Veja a figura 2.4.

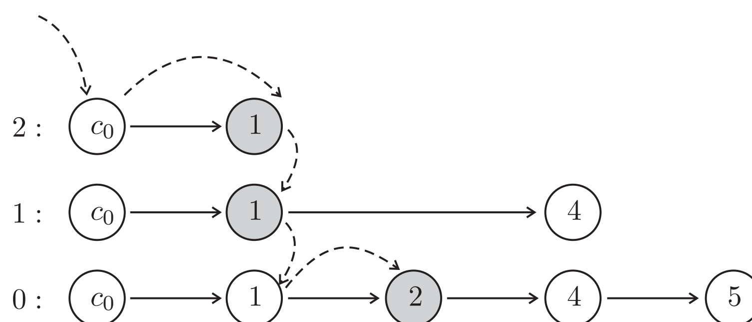

Figura 2.4: *Busca em uma skip list pelo elemento 3, com preenchimento da tabela auxiliar ante, que vai conter referências para as células destacadas nos níveis correspondentes.*

Algoritmo 6 inserir(c_0, e)

1: $ante \leftarrow$ alocarTabela()
2: $c' \leftarrow$ buscarCriaAnte(c_0, e, $ante$)
3: **if** $c'.chave = e$ **then**
4:    **return** ▷ O elemento já está presente
5: $c \leftarrow$ alocarCélula(e, escolherNível())
6: **if** $c.nivel > c_0.nivel$ **then**
7:    **for** $j \leftarrow c_0.nivel + 1$ **to** $c.nivel$ **do**
8:       $c_0.prox[j] \leftarrow \perp$
9:       $ante[j] \leftarrow c_0$
10:    $c_0.nivel \leftarrow c.nivel$
11: **for** $j \leftarrow 0$ **to** $c.nivel$ **do**
12:    $c.prox[j] \leftarrow ante[j].prox[j]$
13:    $ante[j].prox[j] \leftarrow c$

Algoritmo 7 buscarCriaAnte(c, e, $ante$)

1: $j \leftarrow c.nivel$
2: **while** $j \geq 0$ **do**
3:    **if** $c.prox[j] \neq \perp$ **and** $c.prox[j].chave < e$ **then**
4:       **return** buscarCriaAnte($c.prox[j]$, e, $ante$)
5:    **else**
6:       $ante[j] \leftarrow c$
7:       $j \leftarrow j - 1$
8: **if** $c.prox[0] \neq \perp$ **and** $c.prox[0].chave = e$ **then**
9:    **return** $c.prox[0]$
10: **else**
11:    **return** c

Na segunda etapa, entre as linhas 5 e 13 do algoritmo `inserir()`, começamos alocando uma nova célula e inicializando seus valores: seu campo é inicializado com e e seu nível é inicializado aleatoriamente por meio de `escolherNível()`. Caso a nova célula tenha nível pelo menos i, ela será inserida entre $ante[i]$ e $ante[i].prox$. Note que $c_o.nivel$ é atualizado e continua guardando o nível da *skip list*. Veja a figura 2.5.

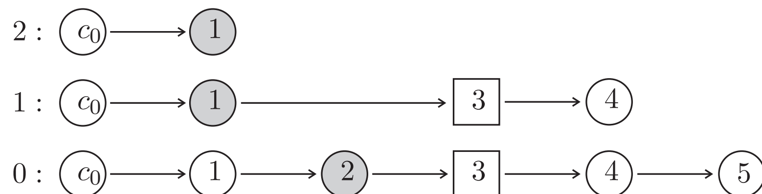

Figura 2.5: *Inclusão da célula 3 na lista ligada da figura 2.4.*

Remoção

A algoritmo 8, `remover(`c_0`, `e`)`, recebe a célula-cabeça c_0 de uma *skip list* e um elemento $e \in U$. Se a *skip list* não contém nenhuma célula cuja chave é e, o algoritmo não modifica a estrutura. Caso contrário, o algoritmo remove da estrutura a célula cuja chave é e e atualiza todas as referências.

A remoção também consiste de duas etapas (veja o algoritmo 8, `remover()`). A primeira etapa, entre as linhas 1 e 4 do algoritmo `remover()`, onde buscamos pelo elemento a ser removido e criamos a tabela auxiliar *ante*, é idêntica à primeira etapa da inserção, utilizando o algoritmo 7, `buscarCriaAnte()`. Veja a figura 2.6.

Na segunda etapa, entre as linhas 5 e 10 do algoritmo `remover(`c_0`, `e`)`, a tabela auxiliar *ante* é usada para remover a célula determinada anteriormente de todas as listas ligadas a que pertence. Veja a figura 2.7. Ao final, usamos a função `liberarCélula()` para liberar essa célula. Na remoção, $c_0.nivel$ é também atualizado quando necessário.

Algoritmo 8 remover(c_0, e)

1: $ante \leftarrow$ alocarTabela()
2: $c \leftarrow$ buscarCriaAnte(c_0, e, $ante$)
3: **if** $c.chave \neq e$ **then**
4:     **return** ▷ O elemento não está presente
5: **for** $j \leftarrow c_0.nivel$ **to** 0 **do**
6:     **if** $ante[j].prox[j] = c$ **then**
7:         $ante[j].prox[j] \leftarrow c.prox[j]$
8: **while** $c_0.nivel > 0$ **and** $c_0.prox[c_0.nivel] = \perp$ **do**
9:     $c_0.nivel \leftarrow c_0.nivel - 1$
10: liberarCélula(c)

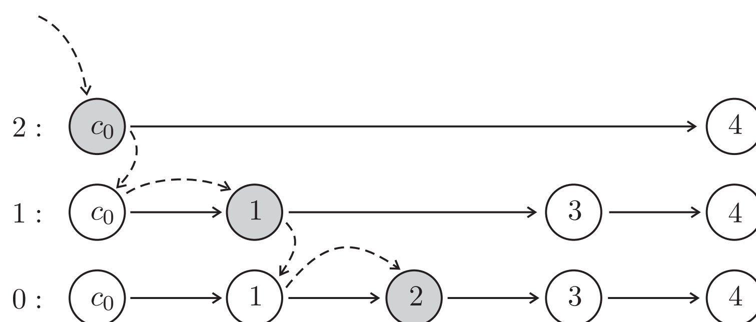

Figura 2.6: *Busca em uma skip list pelo elemento 3, com preenchimento da tabela auxiliar ante, que vai conter referências para as células destacadas nos níveis correspondentes.*

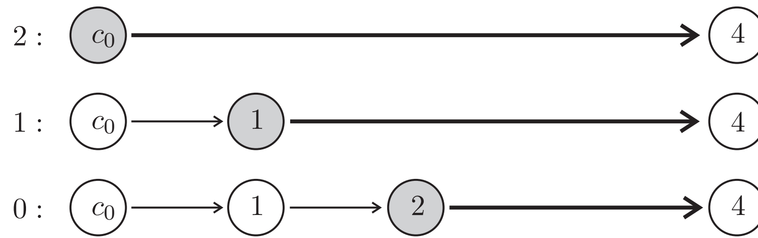

Figura 2.7: *Remoção da célula 3 da lista ligada da figura 2.6.*

2.3 Análise

Nesta seção, vamos fazer uma análise dos algoritmos apresentados. Vamos permitir a existência de *adversários*, ou seja, de "usuários externos" com capacidade de realizar operações sobre a *skip list*, e talvez de forma maliciosa. Porém, vamos assumir que eventuais adversários não tenham acesso aos níveis das células, que foram escolhidos aleatoriamente. Na nossa análise, vamos considerar uma *skip list* com parâmetro p (genérico), cujas células-elemento sejam $c_1, \ldots, c_n$, na ordem que aparecem na lista S_0.

A análise feita nessa seção é baseada na análise de Pugh [29].

2.3.1 Infra-estrutura

O algoritmo `inicializar()` consome claramente tempo constante. Para calcular o tempo consumido no algoritmo `escolherNível()`, consideremos inicialmente uma seqüência de experimentos, onde cada experimento é um evento aleatório independente em que a probabilidade de sucesso seja p e a probabilidade de insucesso seja $1-p$. A variável aleatória $G(p)$, que representa o número de experimentos realizados até o primeiro sucesso, é dita uma variável aleatória com distribuição geométrica de parâmetro p. Temos que

$$E[G(p)] = \sum_{i=1}^{\infty} \Pr[G(p) \geq i] = \sum_{i=1}^{\infty} (1-p)^{i-1} = \sum_{i=0}^{\infty} (1-p)^i = \frac{1}{p}.$$

Voltando ao algoritmo `escolherNível(`p`)`, se assumirmos que `rand()` gasta tempo constante, o algoritmo `escolherNível(`p`)` gasta tempo esperado $O(1+E[G(1-p)])$. Como $E[G(1-p)] = 1/(1-p)$, `escolherNível(`p`)` gasta tempo esperado constante.

2.3.2 Busca

No algoritmo 4, ou vamos executar a linha 4 ou a linha 6, mas nunca ambas. O mesmo ocorre no algoritmo 5, nas linhas 6 e 8, e no algoritmo 7, nas linhas 4 e 6. Na estrutura, os três algoritmos

se comportam de maneira idêntica: ou caminhamos lateralmente para visitar uma célula que ainda não foi visitada, ou descemos de nível.

Portanto, fica fácil perceber que o tempo de execução destes algoritmos é proporcional ao número de células distintas visitadas lateralmente somado ao número de níveis descidos. Comecemos limitando a quantidade de níveis descidos.

Lema 1. *Em uma* skip list *com n elementos, o número de níveis é $O(\lg n)$ com alta probabilidade, e o número esperado de níveis é $O(\lg n)$.*

Demonstração. Seja c uma célula-elemento arbitrária da nossa estrutura e c_0 a célula-cabeça. Como vimos antes, para todo $i \geq 0$, $\Pr[c.nivel \geq i] = p^i$. Supondo que a *skip list* é construída através de sucessivas invocações de `inserir()` e `remover()`, e já que $c_0.nivel$ é o maior nível de todas as células-elemento da *skip list*,

$$\Pr[c_0.nivel \geq i] \ \leq \sum_{c \in \{c_1,\ldots,c_n\}} \Pr[c.nivel \geq i] = \sum_{c \in \{c_1,\ldots,c_n\}} p^i = np^i.$$

Para demonstrar a primeira afirmação do lema, vamos considerar $q = 1/p$ e $i = \lceil 2 \lg_q n \rceil$. Notemos que $q > 1$, e logo

$$\Pr[c_0.nivel \geq i] \ \leq \ np^i = \frac{n}{q^i} = \frac{n}{q^{\lceil 2 \lg_q n \rceil}} \leq \frac{n}{q^{2 \lg_q n}} = \frac{n}{n^2} = \frac{1}{n}.$$

Como $\Pr[c_0.nivel \leq \lceil 2 \lg_c n \rceil - 1] \geq 1 - 1/n$ e $\lceil 2 \lg_c n \rceil - 1 = O(\lg n)$, segue a primeira afirmação do lema.

Para demonstrar a segunda afirmação do lema, vamos aproveitar uma parte do cálculo anterior:

$$\begin{aligned}
E[c_0.nivel] &= \sum_{i=0}^{\infty} \Pr[c_0.nivel \geq i] \\
&= \sum_{i=0}^{\lceil 2\lg_q n\rceil - 1} \Pr[c_0.nivel \geq i] + \sum_{i=\lceil 2\lg_q n\rceil}^{\infty} \Pr[c_0.nivel \geq i] \\
&\leq \lceil 2\lg_q n\rceil + np^{\lceil 2\lg_q n\rceil} \sum_{i=0}^{\infty} p^i \\
&= \lceil 2\lg_q n\rceil + \frac{n}{q^{\lceil 2\lg_q n\rceil}} \frac{1}{1-p} \\
&\leq \lceil 2\lg_q n\rceil + \frac{1}{n(1-p)} \\
&= O(\lg n).
\end{aligned}$$

□

Calculamos a primeira componente do tempo da busca, a quantidade de níveis descidos. Agora vamos calcular a segunda componente, a quantidade de células distintas visitadas lateralmente.

Lema 2. *Em uma busca, o número esperado de células distintas visitadas em cada nível é $O(1)$.*

Demonstração. Em uma busca, quando estamos percorrendo a lista ligada de nível i, e encontramos uma célula de nível $j > i$, tal célula deve vir depois do elemento procurado, ou então a teríamos atingindo enquanto estávamos percorrendo a lista ligada de nível j. Além disso, se essa célula de fato vem depois do elemento procurado, devemos descer de nível. Portanto, na nossa busca, sempre que encontramos uma célula com nível maior que o nível corrente, devemos descer de nível.

A probabilidade de uma célula c, que esteja presente na lista ligada de nível i, estar também presente na lista ligada de nível $i+1$ é p (basta calcular a probabilidade condicional $Pr[c.nivel \geq (i+1) | c.nivel \geq i] = p$). Seja v_i o número de células distintas visitadas na lista ligada de nível i. Portanto, pelo que foi dito no parágrafo anterior,

$$E[v_i] = E[G(1-p)] - 1 = O(1).$$

□

A quantidade esperada de células distintas visitadas lateralmente em todos os níveis é então facilmente calculada:

Corolário 1. *Em uma busca (algoritmos* `buscar()` *ou* `buscar2()`*), o número esperado de células distintas visitadas é* $O(\lg n)$.

Demonstração. Isso segue diretamente dos lemas 1 e 2. □

Somando o número esperado de níveis descidos, $O(\lg n)$ pelo lema 1, com o número esperado de células visitadas lateralmente, $O(\lg n)$ pelo corolário 1, uma busca em uma *skip list* com n elementos presentes gasta tempo esperado $O(\lg n)$.

2.3.3 Inserção

A busca inicial, como vimos, gasta tempo esperado $O(\lg n)$. O tempo consumido nos laços das linhas 7 e 11, podemos perceber, é proporcional à quantidade de níveis da *skip list*. O tempo consumido nas outras linhas é claramente $O(1)$. Portanto, o algoritmo `inserir()` gasta tempo esperado $O(\lg n)$.

2.3.4 Remoção

A análise é similar ao caso anterior, mas devemos considerar os laços das linhas 5 e 8. Da mesma forma, o tempo consumido nesses laços é proporcional à quantidade de níveis da *skip list*. Portanto, o algoritmo `remover()` gasta tempo esperado $O(\lg n)$.

Pugh [29] observa que *skip lists* têm os limites de tempo assintóticos esperados iguais aos de árvores balanceadas de busca, com algoritmos mais simples e com menos consumo de memória.

Capítulo 3

Skip Graphs

Neste capítulo, vamos apresentar uma estrutura de dados denominada *skip graph*. Esta estrutura é baseada nas *skip lists* do capítulo anterior, e também permite realizar buscas, inserções e remoções eficientemente em um conjunto de elementos oriundos de um universo totalmente ordenado U. Ela foi definida por Aspnes e Shah [6], tendo em vista ambientes distribuídos[1]. Nesse capítulo, vamos apresentar inicialmente uma versão dessa estrutura para ambientes tradicionais, ou seja, sistemas com um único processador e um único dispositivo de memória, acessível globalmente. Em capítulos seguintes, vamos apresentar uma versão concorrente tendo em vista ambientes multiprocessados, e também discutir a respeito da adaptação dessa versão para ambientes distribuídos.

Um *skip graph* é uma lista ligada ordenada com atalhos. Porém, em cada nível $i > 0$, existem várias listas ligadas, todas duplamente encadeadas e sem cabeça. Então, em cada nível $i > 0$, há bem mais atalhos do que em uma *skip list*, estabelecidos por múltiplas listas ligadas e em ambas as direções.

Tomemos um σ inteiro, $\sigma > 1$. Informalmente, em um *skip graph* com parâmetro σ, o nível 0 consiste de uma lista ligada com todas as células da estrutura, o nível 1 consiste de até σ listas ligadas que refinam a lista ligada de nível 0, o nível 2 consiste de até σ^2 listas ligadas que refinam as listas ligadas de nível 1, e assim por diante. À medida que os níveis aumentam, temos listas ligadas menores e em maior quantidade. Em cerca de $O(\lg n)$ níveis, se espera que existam n listas ligadas unitárias, onde n é o número de elementos da estrutura. Um exemplo de *skip graph* é apresentado na figura 3.1.

[1] Ambientes distribuídos que empregassem *skip lists* sofreriam bastante com a má distribuição de carga: a cabeça da *skip list* seria um nó que teria um número de acessos significativo.

3.1 Estrutura de Dados

Nesta seção, vamos formalizar a definição de *skip graphs*. Para isso, vamos estabelecer primeiramente alguns conceitos fundamentais.

Seja $\sigma > 1$ um inteiro e $\Sigma = \{0, \ldots, \sigma - 1\}$ um conjunto que vamos chamar de *alfabeto*. *Símbolos* são elementos do alfabeto Σ. *Palavras* sobre Σ são seqüências finitas ou infinitas de elementos de Σ. O conjunto de todas as palavras finitas e o conjunto de todas as palavras infinitas sobre Σ são denominados respectivamente de Σ^* e Σ^∞. O número de símbolos de uma palavra w é chamado de *tamanho* de w e é denotado por $|w|$. A palavra sem nenhum símbolo, de tamanho 0, é denominada ϵ. O conjunto de todas as palavras sobre Σ com tamanho i é denominado Σ^i, para $i \geq 0$. O prefixo de w de tamanho i é denotado por $w \rhd i = w[0\,..\,i-1]$. Se uma palavra w é prefixo de outra palavra z, então escrevemos $w \sqsubset z$.

Os *skip graphs*, como dissemos, são uma lista ligada ordenada com atalhos. Vamos continuar observando tal estrutura sob o ponto de vista de vários níveis de listas ligadas. Em um *skip graph*, a lista ligada de nível 0 é denominada S_ϵ, as listas ligadas de nível 1 são denominadas $S_0, \ldots, S_{(\sigma-1)}$, as listas ligadas de nível 2 são denominadas $S_{00}, \ldots, S_{(\sigma-1)(\sigma-1)}$, e assim por diante. Em geral, as listas ligadas de nível i são denominadas S_w, para todo $w \in \Sigma^i$. Sem exceção, todas elas estão ordenadas e duplamente encadeadas. Esse fato implica nos atalhos serem bidirecionais.

Uma célula qualquer c está associada uma palavra infinita $c.palavra \in \Sigma^\infty$, determinada aleatoriamente. Na prática, vamos manter apenas um certo prefixo finito de cada palavra infinita, com tais símbolos tomados de modo uniforme e independente (como veremos adiante) em Σ. A probabilidade de duas células distintas serem associadas à mesma palavra infinita é 0. Vamos chamar essas diversas palavras de *palavras aleatórias*. As palavras aleatórias indicam em quais listas ligadas uma celula qualquer c vai estar presente. Todas as células estão presentes na lista ligada S_ϵ, que possui nível 0. As células de S_ϵ que têm $palavra \rhd 1 = 0$ vão estar na lista ligada S_0, as que têm $palavra \rhd 1 = 1$ vão estar na lista ligada S_1, e assim por diante. As células de S_0 que têm $palavra \rhd 2 = 00$ vão estar na lista ligada S_{00}, as que têm $palavra \rhd 2 = 01$ vão estar na lista ligada S_{01}, e assim por diante. As listas ligadas vão assim sendo refinadas a cada nível, e esse processo termina quando chegarmos a uma lista ligada *unitária*, que contém apenas uma célula. Naturalmente, as listas ligadas que não possuem nenhuma célula não se manifestam na representação da estrutura em memória. Portanto, um *skip graph* só existe em memória quando não estiver vazio.

As listas ligadas de um *skip graph* não possuem cabeça, logo toda célula vai guardar um elemento do conjunto U. Qualquer célula c está presente em S_w só se $w \sqsubset c.palavra$. Vamos dizer que c tem nível i se a lista ligada unitária que contém c é de nível i. Vamos denotar a chave e o nível de uma célula c como fizemos nas *skip lists*, ou seja, por $c.chave$ e $c.nivel$. Além disso, se c está em S_w, que é

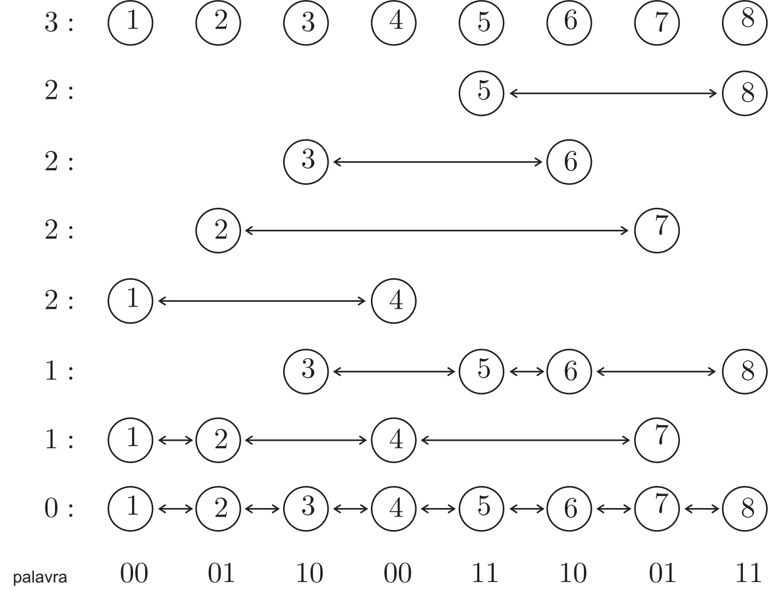

Figura 3.1: *Um exemplo de skip graph com nível 3. Abaixo temos as palavras aleatórias que ditam em que listas ligadas cada célula está presente. Por acaso, as células do exemplo possuem todas o mesmo nível.*

de nível i, seu apontador para a célula seguinte em S_w é denominado $c.prox[i]$, e seu apontador para a célula anterior em S_w é denominado $c.ante[i]$. Esses apontadores são chamados respectivamente de apontador à direita e à esquerda de nível i da célula c. O nível de um *skip graph* é o maior dentre os níveis de todas as células do *skip graph*.

Agora supomos que uma célula c esteja presente em uma lista ligada S_w. Vamos definir a vizinha à direita de c de nível $|w|$ como a célula $d \in S_w$ que vem logo depois de $c.chave$ em S_w, e a vizinha à esquerda de c de nível $|w|$ como a célula $e \in S_w$ que vem logo antes de $c.chave$ em S_w. Esses vizinhos podem também ser chamados, respectivamente, de vizinhos à direita ou à esquerda de c em S_w. Por definição, $c.prox[i]$ e $c.ante[i]$ apontam, respectivamente, para as vizinhas à direita e à esquerda de c na lista ligada de nível i a que pertencem c, $c.ante[i]$ e $c.prox[i]$.

Dizemos que células c_1 e c_2 são *compatíveis* no nível i se $c_1.palavra \triangleright i = c_2.palavra \triangleright i$. Assim, duas células estão presentes na mesma lista ligada de nível i se e somente se estas células são compatíveis no nível i. Finalmente, vamos dizer também que S_{w_1} é uma lista ligada antecedente de S_{w_2} ou que S_{w_2} é uma lista ligada descendente de S_{w_1} se $w_1 \sqsubset w_2$.

Definição 1. *Um* skip graph *com parâmetro σ é uma lista ligada ordenada com atalhos, sem cabeça e duplamente encadeada, onde o nível de uma célula é o menor i tal que o prefixo de i símbolos da palavra aleatória dessa célula seja único dentre todos os prefixos de i símbolos das palavras aleatórias de todas as outras células. Para toda célula c do* skip graph, *o campo $c.chave$ guarda sua chave, o*

campo c.nivel guarda seu nível, o campo c.palavra guarda os primeiros c.nivel símbolos de sua palavra aleatória e o campo c.prox é uma tabela com $c.nivel + 1$ *entradas, onde* $c.prox[i]$ *e* $c.ante[i]$ *apontam para a vizinha à direita e a vizinha à esquerda de* c *de nível* i*, respectivamente.*

Lema 1. *Seja* G *um* skip graph *com parâmetro* σ*, que contém* n *células. Para toda célula* c *de* G*, a coleção* $S_c = \{T_0, \ldots, T_k\}$*, onde* $T_i = S_{c.palavra \rhd i}$ *e* $k = c.nivel$*, é uma* skip list *com parâmetro* $p = 1/\sigma$*, mas duplamente encadeada e sem cabeça alguma.*

Demonstração. Por definição, $T_0 = S_\epsilon$, e a probabilidade de que qualquer célula de G esteja presente em T_0 é 1.

Considere uma célula $x \in G$ arbitrária. Vamos dizer que o nível de x em S_c é o maior i tal que $0 \leq i \leq k$ e $x \in T_i$. Como $x \in T_0$, o nível de x em S_c é ≥ 0. Se x está presente em T_i, então temos que $x.palavra \rhd i = c.palavra \rhd i$ e o nível de x em S_c é $\geq i$.

Vamos supor que $x \in T_i$. Dessa forma, já que os símbolos das palavras aleatórias são escolhidos independente e uniformemente no momento da inserção, a probabilidade de que x esteja presente em T_{i+1} é a probabilidade de que o $(i + 1)$-ésimo símbolo de *x.palavra* e *c.palavra* sejam iguais, o que sabemos ser igual a $p = 1/|\Sigma| = 1/\sigma$. Já podemos concluir que o nível de x em S_c é também determinado independente e uniformemente, pois depende somente dos símbolos presentes nas palavras aleatórias.

Falta mostrar ainda que $\Pr[\text{nível de } x \text{ em } S_c = j] = p^j(1 - p)$. Pelo que vimos acima, podemos concluir facilmente que $\Pr[\text{nível de } x \text{ em } S_c \geq j] = p^j$. Para x ter nível exatamente j em S_c, x não pode ter nível $j + 1$ em S_c, o que ocorre com probabilidade $1 - p$. Desta forma, podemos concluir que $\Pr[\text{nível de } x \text{ em } S_c = j] = p^j(1 - p)$. □

Sendo assim, de agora em diante, vamos nos referir a S_c como a *skip list* de c. Notemos, portanto, que um *skip graph* é composto de diversas *skip lists*, uma para cada célula do *skip graph*, e que tais *skip lists* compartilham seus níveis inferiores. Veja a figura 3.2.

3.2 Algoritmos

Nesta seção, vamos apresentar os algoritmos empregados em um *skip graph* para realizar buscas, inserções e remoções.

Infra-estrutura

Em qualquer célula c, *c.nivel* deve ser sempre igual ao índice da última entrada das tabelas *ante* e *prox*. Desta forma, quando mudarmos o nível da célula, novas entradas devem ser adicionadas

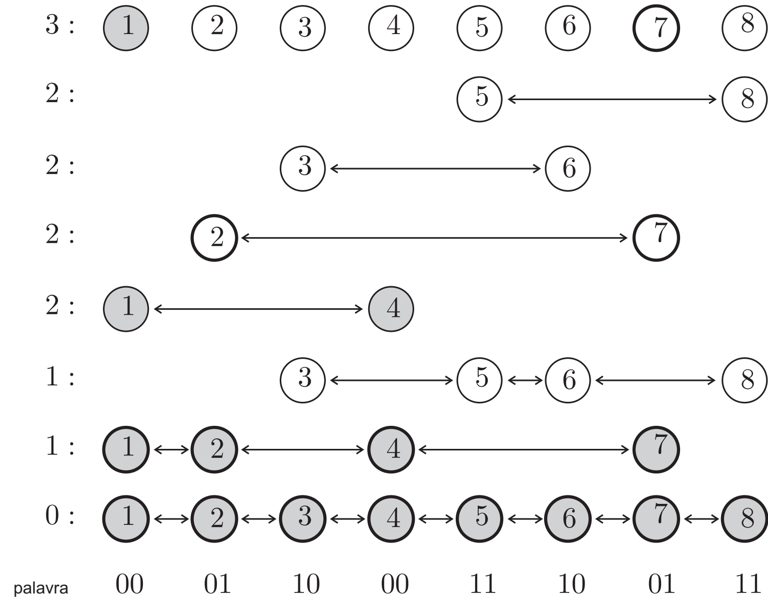

Figura 3.2: *Compartilhamento de níveis em um skip graph: as skip lists S_1 e S_7 compartilham os dois níveis inferiores.*

ou removidas no final dessas tabelas, mantendo o invariante de que *c.nivel* seja sempre igual ao índice da última entrada das tabelas *ante* e *prox*. Além disso, como usaremos somente os *c.nivel* primeiros símbolos de *c.palavra* nos nossos algoritmos, vamos armazenar somente os *c.nivel* primeiros símbolos de *c.palavra*, que então serão gerados sob demanda: ao incrementar o nível de uma célula, um novo símbolo s deverá ser escolhido, independente e uniformemente distribuído em $\{0, \dots, \sigma - 1\}$, e adicionado ao final de sua palavra aleatória; ao decrementar o nível de uma célula, o último símbolo de sua palavra aleatória deverá ser destruído. A escolha do símbolo a ser adicionado será feita sempre usando a rotina `rand()` do capítulo anterior.

O algoritmo 9, `alocarCélula(`e`, `j`)`, permite alocar uma nova célula c para o *skip graph*, atribuindo $e \in U$ ao campo *c.chave* e ajustando os outros campos de tal forma que o nível da célula seja $j \geq 0$. A função `alocarTabela()` aloca e devolve uma tabela vazia, conforme definimos no capítulo anterior. O algoritmo 10, `atribuirNível(`c`, `j'`)`, recebe uma célula $c \neq \bot$ e um nível $j' \geq 0$, e ajusta os campos da célula c de forma que seu nível seja j'. Os ajustes são feitos conforme discutimos no parágrafo anterior.

O algoritmo 11, `atribuirVizinho(`c`, `c'`, `d`, `j`)`, recebe uma célula $c \neq \bot$, uma célula c' (essa já pode ser $\bot$), uma direção $d \in \{D, E\}$ e um nível $j \geq 0$, e faz com que c tenha c' como novo vizinho à direita (se $d = D$) ou à esquerda (se $d = E$), no nível j. A célula c, e também a célula c' (se $c' \neq \bot$), já devem estar presentes na estrutura e ter nível $\geq j$ na invocação do algoritmo. Dependendo da

Algoritmo 9 alocarCélula(e, j)

```
1: ▷ crie c, uma nova célula alocada na memória
2: c.chave ← e
3: c.nivel ← −1
4: c.ante ← alocarTabela()
5: c.prox ← alocarTabela()
6: c.palavra ←
7: atribuirNível(c, j)
8: return c
```

Algoritmo 10 atribuirNível(c, j')

```
1: j1 ← c.nivel
2: j2 ← j'
3: if j1 < j2 then
4:    for j ← j1 + 1 to j2 do
5:       ▷ Criar entradas c.ante[j], c.prox[j]
6:       c.ante[j] ← ⊥
7:       c.prox[j] ← ⊥
8:       if j > 0 then
9:          ▷ Criar c.palavra[j − 1]
10:         c.palavra[j − 1] ← rand(σ)
11: else
12:    for j ← j2 down to j1 + 1 do
13:       ▷ Destruir entradas c.ante[j], c.prox[j]
14:       if j > 0 then
15:          ▷ Destruir c.palavra[j − 1]
16: c.nivel ← j'
```

mudança executada, o nível de c pode precisar de certos ajustes: se c ganhou algum vizinho $\neq \perp$ na sua lista ligada unitária, c deve subir de nível (algoritmo 11, linha 6); se c perdeu seu último vizinho $\neq \perp$ na sua lista ligada unitária, c deve descer de nível (algoritmo 11, linha 8). Os ajustes necessários são todos feitos pelo algoritmo. Veja as figuras 3.3 e 3.4.

Busca

A busca em um *skip graph* é bastante parecida com a busca em uma *skip list*, mas agora usamos o algoritmo 12, `buscar(`c`, `j`, `e`)`, que recebe uma célula qualquer $c \neq \perp$, um nível j e um elemento $e \in U$, e devolve a célula que vem logo antes de e incluindo e se $c.chave < e$, a célula que vem logo depois de e incluindo e se $c.chave > e$ ou, trivialmente, a célula c, se $c.chave = e$. Tipicamente, o algoritmo é invocado com $j = c.nivel$. Podemos perceber no algoritmo que ele caminha à direita ou

Algoritmo 11 atribuirVizinho(c, c', d, j)

1: **if** $d = D$ **then**
2: $\quad c.prox[j] \leftarrow c'$
3: **else if** $d = E$ **then**
4: $\quad c.ante[j] \leftarrow c'$
5: **if** $c.ante[c.nivel] \neq \bot$ **and** $c.prox[c.nivel] \neq \bot$ **then**
6: $\quad$ atribuirNível(c, $c.nivel + 1$)
7: **else if** $c.nivel > 0$ **and** $c.ante[c.nivel - 1] = \bot$ **and** $c.prox[c.nivel - 1] = \bot$ **then**
8: $\quad$ atribuirNível(c, $c.nivel - 1$)

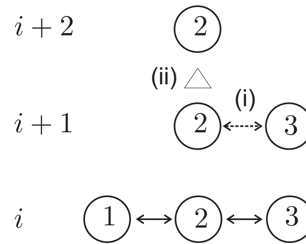

Figura 3.3: *Criação de um novo nível no skip graph: ao ganhar a vizinha* 3 *no nível* $i + 1$ *(i), o nível de* 2 *é incrementado (ii).*

à esquerda, dependendo do ponto em que se iniciou a busca. Assim, não precisamos de cabeça, mas precisamos de listas duplamente encadeadas.

No algoritmo 12, `buscar()`, temos sempre o seguinte:

1. $c \neq \bot$ e $c.nivel \geq j$;

2. se começamos a busca em uma célula com chave $\leq e$, temos sempre $c.chave \leq e$; caso contrário, temos sempre $c.chave \geq e$.

Notemos que a segunda observação feita anteriormente garante que qualquer apontador devol-

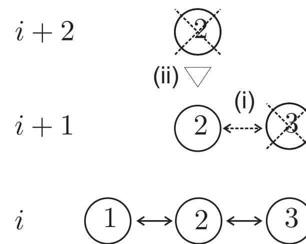

Figura 3.4: *Remoção de um nível no skip graph: ao perder a vizinha* 3 *no nível* $i + 1$ *(i), o nível de* 2 *é decrementado (ii).*

Algoritmo 12 buscar(c, j, e)

```
1:  if c.chave < e then
2:    if c.prox[j] ≠ ⊥ and c.prox[j].chave ≤ e then
3:      return buscar(c.prox[j], j, e)
4:    else
5:      if j ≥ 1 then
6:        return buscar(c, j − 1, e)
7:      else
8:        return c
9:  else if c.chave > e then
10:   if c.ante[j] ≠ ⊥ and c.ante[j].chave ≥ e then
11:     return buscar(c.ante[j], j, e)
12:   else
13:     if j ≥ 1 then
14:       return buscar(c, j − 1, e)
15:     else
16:       return c
17: return c
```

vido por uma busca seja $\neq \perp$. Logo adiante, a abordagem recursiva de escrita do algoritmo será devidamente justificada.

Considere uma busca em um *skip graph*, que foi iniciada em c. Nesta busca, quando estamos visitando uma célula x no nível j, ou vamos descer para o nível $j-1$ ou vamos visitar um dos vizinhos de nível j de x. Por definição, os vizinhos de x são compatíveis com x, portanto a célula que estamos visitando é sempre uma célula de S_c, desde o início. Além disso, a busca começa do maior nível em S_c, já que c possui o maior nível dentre todos em S_c, e caminha somente em um sentido, conforme as linhas 1-3 e 9-11. Assim temos que as buscas em *skip graphs* não passam de buscas em *skip lists*. Por esse motivo o algoritmo é escrito de maneira recursiva, já que *skip lists* têm um caráter recursivo (conforme discutimos no capítulo anterior). Além disso, sob tal forma, fica mais fácil perceber a semelhança que existe entre os dois algoritmos. Veja a figura 3.5.

Inserção

O algoritmo 13, `inserir(a, e)`, é usado na inserção de elementos no *skip graph*. Ele recebe uma célula a e um elemento $e \in U$. Se o *skip graph* já contém uma célula com chave e, o algoritmo não modifica a estrutura. Caso contrário, o algoritmo insere na posição correta da estrutura uma nova célula cuja chave é e e atualiza todas as referências. O parâmetro a é uma célula qualquer do *skip graph*, denominada *apresentador*. Se $a = \perp$, o algoritmo 13 cria um novo *skip graph* contendo uma única célula, cuja chave é e. Caso contrário, a vai ser usada para verificar se o elemento e já está

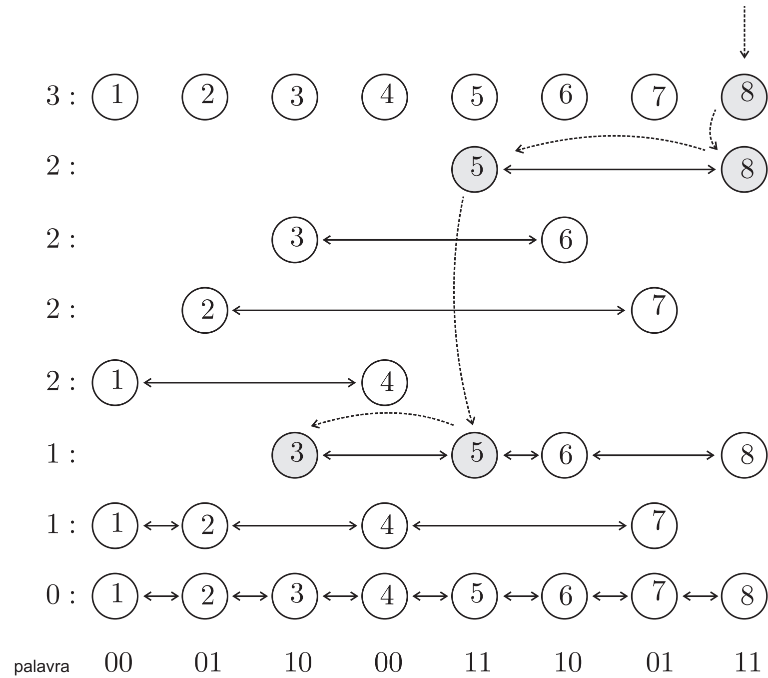

Figura 3.5: *Busca pelo elemento* 3 *em um skip graph.*

presente no *skip graph*, e, se não estiver, para encontrar a posição em S_ϵ onde será inserida a nova célula com chave e.

A inserção de uma nova célula c em um *skip graph* é feita de maneira indutiva: c é inserida no nível 0, e então, com c já inserida no nível i, vamos inserir c no nível $i+1$. Em um primeiro momento, encontramos as duas células que serão vizinhas de c no nível 0 e inserimos c entre essas duas células, no nível 0. Em um segundo momento, repetimos as seguintes operações (veja a figura 3.6): (1) tomamos o último nível i em que já inserimos c; (2) em cada direção, vamos percorrer as células vizinhas de c no nível i, a fim de encontrar as células que serão tornadas vizinhas de c no nível $i+1$; e (3) inserimos c no nível $i+1$, entre as células encontradas anteriormente. O algoritmo 14, `encontrarVizinhoCompatAcima()`, é usado para procurar por células vizinhas compatíveis nos níveis superiores; já o algoritmo 11, `atribuirVizinho()`, é usado para atribuir vizinhos.

O algoritmo 14, `encontrarVizinhoCompatAcima(`c'`, `c''`, `d`, `j`)`, recebe uma célula c' (eventualmente $\perp$), uma célula $c'' \neq \perp$, uma direção $d \in \{D, E\}$ e um nível $j \geq 0$, e devolve a primeira célula compatível com c' que vem antes de $c'.chave$, se $d = E$, ou que vem depois de $c'.chave$, se $d = D$, dentre as células da lista ligada de nível j à qual pertence c''. Se tal célula não existir, o algoritmo devolve $\perp$. Na invocação do algoritmo, se $c' \neq \perp$ e $c'' \neq \perp$, c' e c'' devem ser compatíveis no nível j. O algoritmo, a partir de c', visita células na direção d e no nível j, procurando por uma célula

Algoritmo 13 inserir(a, e)

```
1:  c ← alocarCelula(e, −1)
2:  if a = ⊥ then
3:     return ▷ Primeiro elemento: não há mais nada a fazer
4:  c′ ← buscar(a, a.nivel, e)
5:  if c′.chave = e then
6:     liberarCelula(c)
7:     return ▷ O elemento já está presente
8:  else if c′.chave < e then
9:     c_e ← c′
10:    c_d ← c_e.prox[0]
11: else if c′.chave > e then
12:    c_d ← c′
13:    c_e ← c_d.ante[0]
14: atribuirVizinho(c, c_e, E, 0)
15: atribuirVizinho(c, c_d, D, 0)
16: if c_e ≠ ⊥ then
17:    atribuirVizinho(c_e, c, D, 0)
18: if c_d ≠ ⊥ then
19:    atribuirVizinho(c_d, c, E, 0)
20: j ← 0
21: while true do
22:    c_e ← encontrarVizinhoCompatAcima(c.prox[j], c, E, j)
23:    c_d ← encontrarVizinhoCompatAcima(c.ante[j], c, D, j)
24:    if c_e = ⊥ and c_d = ⊥ then
25:       break
26:    atribuirVizinho(c, c_e, E, j + 1)
27:    atribuirVizinho(c, c_d, D, j + 1)
28:    if c_e ≠ ⊥ then
29:       atribuirVizinho(c_e, c, D, j + 1)
30:    if c_d ≠ ⊥ then
31:       atribuirVizinho(c_d, c, E, j + 1)
32:    j ← j + 1
```

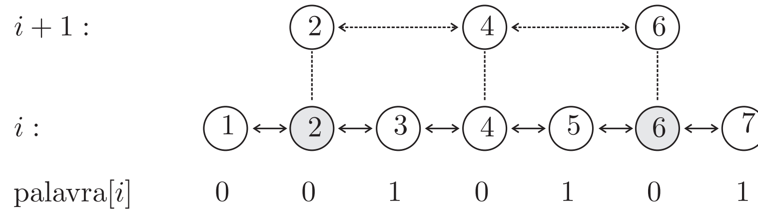

Figura 3.6: *Inserção da célula 4 em um nível $i+1$ após a inserção no nível i: as células compatíveis em ambos os lados são encontradas, e a nova célula é colocada entre elas. Abaixo, mostramos o símbolo da palavra aleatória usado na verificação de compatibilidade.*

compatível com c'' no nível $j+1$. Como c' e c'' são compatíveis no nível j, os primeiros j símbolos de suas palavras aleatórias são idênticos, portanto só precisamos comparar o $(j+1)$-ésimo símbolo de suas palavras aleatórias para verificar a compatibilidade de c' e c'' no nível $j+1$ (é importante ressaltar que o $(j+1)$-ésimo símbolo da palavra tem índice j, lembrando que começamos a contar a partir do índice 0).

O algoritmo é escrito da maneira recursiva pelo seguinte: se c' e c'' forem compatíveis, podemos devolver c'; em caso contrário, o vizinho à esquerda ou à direita de c', conforme a direção d, é o próximo candidato a ser compatível com c''. Assim, é simples escrever o algoritmo de maneira sucinta (veja o algoritmo 14).

Algoritmo 14 encontrarVizinhoCompatAcima(c', c'', d, j)

1: **if** $c' = \perp$ **or** $c'.palavra[j] = c''.palavra[j]$ **then**
2:     **return** c'
3: **if** $d = E$ **then**
4:     **return** encontrarVizinhoCompatAcima($c'.prox[j]$, c'', d, j)
5: **else if** $d = D$ **then**
6:     **return** encontrarVizinhoCompatAcima($c'.ante[j]$, c'', d, j)

Remoção

O algoritmo 15, `remover(`a`,` e`)`, é usado na remoção de elementos do *skip graph*. Ele recebe uma célula a e um elemento $e \in U$. Se o *skip graph* não contém nenhuma célula cuja chave é e, o algoritmo não modifica a estrutura. Caso contrário, o algoritmo remove da estrutura a célula cuja chave é e e atualiza todas as referências. O parâmetro a, a célula apresentadora, vai ser usada para verificar se o elemento e já está presente no *skip graph*, e, se estiver, para encontrar a posição em S_ϵ de onde será removida a célula com chave e. Note que esse procedimento exclui totalmente a *skip list* da célula com chave e do *skip graph*.

Algoritmo 15 remover(a, e)

```
1: c ← buscar(a, a.nivel, e)
2: if c.chave = e then
3:   for j ← c.nivel down to 0 do
4:     c_e ← c.ante[j]
5:     c_d ← c.prox[j]
6:     if c_e ≠ ⊥ then
7:       atribuirVizinho(c_e, c_d, D, j)
8:     if c_d ≠ ⊥ then
9:       atribuirVizinho(c_d, c_e, E, j)
10:  liberarCelula(c)
```

3.3 Análise

Nessa seção, vamos fazer a análise dos algoritmos apresentados. Vamos considerar que eventuais adversários não tenham acesso às palavras aleatórias das células. Conseqüentemente, não têm acesso ao nível das células.

3.3.1 Busca

Como vimos na seção 3.2, as buscas em *skip graphs* não passam de buscas em *skip lists*, o que permite concluir diretamente o tempo gasto nessas operações:

Corolário 1. *Uma busca em um* skip graph *com n células gasta tempo esperado $O(\lg n)$.*

3.3.2 Inserção

No decorrer da inserção, usamos os algoritmos 11 e 14. O algoritmo 11 gasta tempo $O(1)$. O algoritmo 14 percorre células c até que $c = \perp$ ou $c.palavra[j] = c'.palavra[j]$. Como pior caso, consideremos que a primeira situação nunca ocorra, ou seja, que sempre existam mais células para testarmos a compatibilidade. Para cada $c \neq \perp$, a probabilidade de que $c.palavra[j] = c'.palavra[j]$ é igual a $1/\sigma$, já que os símbolos de qualquer palavra aleatória são escolhidos independente e uniformemente do conjunto $\{0, \ldots, \sigma - 1\}$. Assim, o número esperado de invocações recursivas seria igual a $O(E[G(1/\sigma)]) = O(\sigma)$. Considerando que σ é uma constante, o algoritmo 14 gasta tempo esperado $O(1)$.

Lema 2. *Uma inserção em um* skip graph *com n células gasta tempo esperado $O(\lg n)$.*

Demonstração. Na busca inicial, vamos gastar tempo esperado $O(\lg n)$. Para inserir a nova célula c no nível 0, nas linhas 14-19, gastamos tempo $O(1)$, já que o algoritmo 11 é $O(1)$. Para inserir c nos

níveis superiores, no laço da linha 21, gastamos tempo esperado $O(1)$ em cada iteração, já que o algoritmo 11 gasta tempo $O(1)$ e o algoritmo 14 gasta tempo esperado $O(1)$. A quantidade de níveis em que inserimos c é igual ao nível de S_c, que possui valor esperado $O(\lg n)$ (capítulo 2, lema 1). Desta forma, o algoritmo `inserir()` gasta tempo esperado $O(\lg n)$. □

3.3.3 Remoção

A respeito da remoção, temos:

Lema 3. *Uma remoção em um* skip graph *com n células gasta tempo esperado $O(\lg n)$.*

Demonstração. Na busca inicial, vamos gastar tempo esperado $O(\lg n)$. No laço da linha 3, gastamos tempo $O(1)$ em cada iteração, já que o algoritmo 11 é $O(1)$. A quantidade de níveis dos quais temos que remover uma célula é limitada pelo nível da *skip list* dessa célula no *skip graph*, que possui valor esperado $O(\lg n)$ (capítulo 2, lema 1). Portanto, no laço da linha 3, vamos gastar tempo esperado $O(\lg n)$. Desta forma, o algoritmo `remover()` gasta tempo esperado $O(\lg n)$. □

Os *skip graphs* permitem realizar operações de inserção, remoção e busca a um custo assintótico semelhante ao custo assintótico associado às *skip lists*, e além disso permitem que as buscas se iniciem a partir de qualquer célula. Isso é uma característica importante em sistemas distribuídos ou mesmo em ambientes concorrentes onde os tempos de acesso à memória principal não são idênticos para os diversos processadores (como em sistemas multiprocessados NUMA).

Capítulo 4

Skip Graphs Concorrentes

Em sistemas *multithreaded*, multiprocessados ou distribuídos, temos diversas entidades, sejam *threads* executando em um mesmo processador ou em diferentes processadores, ou então processos executando em diferentes máquinas, que acessam, ao mesmo tempo, recursos compartilhados. Dentre os possíveis recursos, temos dados ou até mesmo estruturas de dados. Tomando o ponto de vista do modo de acesso aos recursos compartilhados, as entidades anteriores se comportam de maneira semelhante na prática: elas realizam operações concorrentes sobre os recursos compartilhados. Vamos abstrair os tipos de entidade por meio de um único termo: *processos*.

Quando temos recursos compartilhados, precisamos controlar a concorrência das operações realizadas sobre eles, ou então a interferência entre essas operações poderia deixar os recursos em questão em um estado inválido. Muitas vezes, o estado final do recurso é um estado que sequer seria atingido em uma execução seqüencial das operações concorrentes realizadas, em alguma ordem arbitrária. Veremos adiante um exemplo desse cenário.

Neste capítulo, vamos descrever uma implementação concorrente dos *skip graphs*. Teremos diversos processadores acessando simultaneamente um único dispositivo de memória. Vamos fazer com que os *skip graphs*, se compartilhados e sujeitos a operações concorrentes feitas pelos diversos processadores em questão, se mantenham consistentes, de acordo com a definição da estrutura. Para isso, precisamos modificar os algoritmos que operam sobre nossa estrutura. Quando fazemos isso, dizemos que tornamos os algoritmos *concorrentes*.

4.1 Modelo

Um dos modelos adotados para lidar com estruturas de dados concorrentes é o PRAM (*Parallel Random Access Machine*) [23,33], e ele será empregado para modelar nosso sistema multiprocessado. Esse é um modelo bem forte, no sentido de que abstrai muitos elementos envolvidos, como o *overhead* de acesso à memória compartilhada. Isso nos permite descrever algoritmos sem que nos preocupemos

com esses detalhes, mas sim com as técnicas mais relevantes da programação concorrente.

No modelo PRAM, temos um número arbitrário de processos e de células de memória compartilhada. As instruções de uma máquina PRAM são divididas em 3 etapas:

1. Leitura de uma célula de memória compartilhada;
2. Operações computacionais;
3. Escrita em uma célula de memória compartilhada.

Os processos podem acessar qualquer célula de memória em um tempo unitário. Cada etapa das instruções é feita também em um tempo unitário, e além disso essas etapas são executadas sincronizadamente em todos os processos[1]. Se uma das etapas acima não convém a uma instrução particular, ela não é realizada, mas seu tempo continua sendo consumido. Por exemplo, uma operação de leitura simples seria totalmente realizável na primeira etapa da instrução, e nas outras etapas nada seria feito.

A única forma pela qual os processos podem trocar informações é através da leitura e escrita em células de memória compartilhada. O resultado de um acesso simultâneo a uma célula de memória compartilhada está sujeito a uma das seguintes políticas:

Leitura exclusiva e escrita exclusiva: O acesso a uma célula de memória é permitido a um único processo por vez;

Leitura concorrente e escrita exclusiva: A leitura em uma célula de memória pode ser feita por vários processos ao mesmo tempo, mas a escrita em uma célula de memória é permitida a um único processo por vez;

Leitura concorrente e escrita concorrente: Tanto a leitura quanto a escrita em uma célula de memória podem ser feitas por vários processos ao mesmo tempo. Quando vários processos realizam operações simultâneas de escrita em uma célula de memória, 3 outras sub-políticas podem ser adotadas:

Arbitrária: O resultado final na célula de memória corresponde a uma operação qualquer dentre as escritas simultâneas;

Prioritária: A cada processo é atribuída uma *prioridade*, e o resultado final na célula de memória é o valor escrito pelo processo de mais alta prioridade;

[1] Todos fazem a etapa 1 simultaneamente, depois todos fazem a etapa 2 simultaneamente e depois todos fazem a etapa 3 simultaneamente.

Consensual: Impomos que os processos que realizem escritas simultâneas nas células de memória as façam com valores todos iguais. Assim, o resultado é obviamente o único valor escrito na célula em questão.

No nosso texto, vamos considerar que o sistema permite *leituras e escritas concorrentes*, considerando que uma célula de memória compartilhada que sofre escritas simultâneas fica sujeita a guardar um valor final *arbitrário* dentre os valores escritos. No entanto, nos nossos algoritmos, vamos fazer com que as operações em memória sejam controladas através de operações externas de sincronização, visando permitir a concorrência entre diversos processos.

Há uma variante do modelo PRAM que admite uma assincronia entre os diversos processos e elimina a hipótese do tempo unitário no acesso à memória compartilhada. A adoção deste modelo no nosso texto, o modelo PRAM assíncrono [13], traria junto uma complexidade desnecessária à discussão principal, e portanto optamos simplesmente pelo modelo PRAM (síncrono).

4.2 Sincronização

Tendo descrito nosso modelo, vamos mostrar os mecanismos de controle que vamos empregar nos algoritmos concorrentes do *skip graph*. As premissas do modelo PRAM, mesmo que adotássemos uma política de leituras e escritas exclusivas, não são suficientes para que simplesmente utilizemos os algoritmos tradicionais em um ambiente multiprocessado. O que ocorre é que, apesar do controle de acesso à memória, as operações realizadas em uma estrutura de dados compartilhada poderiam interferir umas sobre as outras. Por exemplo, enquanto uma célula é usada como apresentador, ela não pode ser removida, ou então em algum instante um acesso inválido poderia ser feito à memória. Outro exemplo é quando uma célula está sendo removida: não podemos simplesmente interligar as vizinhas da célula removida nos níveis a que ela pertence, já que as vizinhas da célula removida podem estar também sendo removidas. Ignorar a concorrência poderia deixar a estrutura em um estado incompatível com sua definição. A interferência entre as operações deve ser controlada por mecanismos explícitos de sincronização. Esse é um problema clássico das disciplinas de programação paralela e sistemas operacionais. No nosso texto, vamos tentar observá-lo sob um ponto de vista mais intuitivo e informal. Tratamentos mais formais para esse problema são devidamente discutidos por Lamport [24, 25].

Locks

Os *locks* são os mecanismos de sincronização mais comuns para algoritmos concorrentes. Eles permitem delimitar, nos algoritmos a serem usados em ambientes concorrentes, "regiões críticas" do código que acessam certos recursos compartilhados. Essas regiões críticas do código, se fossem

executadas simultaneamente por múltiplos processos, poderiam implicar em inconsistências no estado final dos recursos acessados no código em questão. Na verdade, além de *delimitado*, o acesso às regiões críticas do código é também *controlado* pelos *locks*, garantindo que um único processo execute por vez os comandos do código das regiões críticas para cada recurso.

Exemplo

Consideremos um dado compartilhado x. Vamos supor que o valor inicial de x é 0, e que temos dois algoritmos que podem ser executados sobre x: `modificar1()` e `modificar2()` (algoritmos 16 e 17). Como vemos, se executarmos primeiro `modificar1()` e depois `modificar2()`, o valor final de x seria 100. Em contrapartida, se executarmos primeiro `modificar2()` e depois `modificar1()`, esse valor já seria 1000.

Algoritmo 16 modificar1(x)

1: **if** $x < 100$ **then**
2: $\quad x \leftarrow x + 100$

Algoritmo 17 modificar2(x)

1: **if** $x < 100$ **then**
2: $\quad x \leftarrow x + 1000$

Porém, vamos supor que agora temos dois processos A e B, que executam concorrentemente os dois algoritmos. O processo A executa `modificar1()` e o processo B executa `modificar2()`. O seguinte cenário poderia ocorrer:

1. O processo A testa se $x < 100$;
2. O resultado do teste em A é verdadeiro;
3. O processo B testa se $x < 100$;
4. O resultado do teste em B é verdadeiro;
5. O processo A incrementa o valor de x, fazendo $x = 100$;
6. O processo B incrementa o valor de x, fazendo $x = 1100$.

Note que, dependendo de como as operações individuais de `modificar1()` e `modificar2()` se intercalassem, x poderia terminar com um valor que nunca seria atingido em uma execução seqüencial desses algoritmos. Isso é uma inconsistência, e a queremos proibir. Queremos fazer com que o valor

final de x seja um valor que pudesse ser obtido em uma execução seqüencial arbitrária dos algoritmos, no caso 100 ou 1000. Portanto, empregamos mecanismos extras para neutralizar os efeitos colaterais da concorrência, ao mesmo tempo em que se continue permitindo ao máximo a execução concorrente das operações individuais.

Especificamente, o problema do exemplo anterior é que os comandos das linhas 1 e 2 do algoritmo `modificar1()`, e das linhas 1 e 2 do algoritmo `modificar2()`, dizem respeito ao mesmo recurso (o dado x), fazendo operações não atômicas sobre esse dado. Portanto, essas linhas são regiões críticas, e devem ser executadas de forma mutuamente exclusiva entre os processos A e B. Na prática, isso faz com que essas linhas funcionem como um único comando atômico.

Funcionamento

Entram agora em cena os *locks*. Os *locks* permitem que delimitemos as linhas destacadas anteriormente como regiões críticas, que um único processo pode executar por vez. Em termos gerais, os *locks* são associados aos recursos compartilhados nos quais queremos executar operações concorrentes. Eles podem então ser "obtidos" ou "liberados" pelos processos que precisam operar sobre os recursos compartilhados em questão. Apenas um processo pode obter um *lock* por vez. Entre os instantes de obtenção e de liberação de um *lock* l por um processo P, vamos dizer que "P possui l" ou que "l foi obtido por P". Se um processo tenta obter um *lock* de posse de outro processo, ele fica bloqueado até que o processo que possui o *lock* o libere. Se vários processos ficarem bloqueados esperando um *lock*, no momento de sua liberação um único processo, qualquer, será desbloqueado[2].

Voltando ao nosso exemplo, vamos criar um *lock* associado ao dado x, denominado l_x. Vamos modificar nossos algoritmos anteriores para que fiquem como os algoritmos 18 e 19, `modificarConc1()` e `modificarConc2()`, mostrados adiante, respectivamente. Veja que as linhas 1 e 2 dos algoritmos anteriores agora estão delimitadas pelo *lock* de x. Assim, dois processos nunca vão executar ao mesmo tempo a região crítica delimitada, operando ao mesmo tempo sobre o dado x. Supondo que dois processos executem os algoritmos simultaneamente, sobre um mesmo x, o primeiro processo que executar a região crítica vai encontrar o valor de x necessariamente igual a 0. Ele então vai aumentar o valor de x em 100 ou 1000, dependendo do algoritmo associado ao processo considerado. O segundo processo que executar a região crítica já vai encontrar o valor de x maior ou igual a 100, e o valor da variável não será mais modificado. O *lock* permite, notemos, que os comandos presentes em `modificarConc1()` e `modificarConc2()` sejam ainda executados intercaladamente se eles operarem sobre diferentes recursos x. Por isso dizemos que as regiões críticas estão associadas a determinados recursos.

[2]Existem *locks* que respeitam a ordem das tentativas de obtenção no momento de se definir o processo a obtê-lo. Eles são chamados de *locks justos*.

Algoritmo 18 modificarConc1(x)

1: obterLock(l_x) ▷ Rotina atômica
2: **if** $x < 100$ **then**
3: $x \leftarrow x + 100$
4: liberarLock(l_x) ▷ Rotina atômica

Algoritmo 19 modificarConc2(x)

1: obterLock(l_x) ▷ Rotina atômica
2: **if** $x < 100$ **then**
3: $x \leftarrow x + 1000$
4: liberarLock(l_x) ▷ Rotina atômica

Na verdade, se tivermos vários recursos compartilhados, podemos associar um único *lock* a todos os recursos, obtendo-o sempre que fôssemos acessar *algum* deles, ou então associar um *lock* a cada recurso, obtendo somente os *locks* dos recursos que fôssemos acessar. Desta forma, um único *lock* pode "proteger" um ou vários recursos. De qualquer maneira, é preciso que os *locks* sejam respeitados globalmente, isto é, qualquer acesso a um recurso protegido por um certo *lock* deve ser feito sempre com sua prévia obtenção, independentemente do *lock* estar associado a um recurso individual ou a múltiplos recursos. Se um processo modificar recursos diretamente, sem considerar eventuais *locks* que precisam ser obtidos antes da mudança em questão, essa operação pode ser feita ao mesmo tempo que uma modificação que respeitou estritamente o protocolo de obtenção de *locks*, sem haver qualquer "registro" de que ela foi feita.

Quando temos um grande conjunto de recursos compartilhados e um conjunto de algoritmos que lêem ou modificam poucos dados ao mesmo tempo, o tamanho das regiões críticas delimitadas não é normalmente o fator crucial para o desempenho das operações concorrentes. O fator crucial é a disputa entre os diversos processos para obter *locks*. Naturalmente, essa disputa diminui conforme os *locks* sejam mais específicos, ou seja, estejam associados a menos recursos compartilhados. Porém, existem vários problemas que podem ocorrer na obtenção de múltiplos *locks* (como veremos na seção 4.2).

Para exemplificar, vamos supor que temos uma estrutura de dados compartilhada, composta de diversas células, na qual qualquer operação esteja delimitada *completamente* por regiões críticas. Se houvesse somente um *lock*, o *lock* da estrutura completa, um único algoritmo por vez seria permitido. Em contrapartida, se houvesse vários *locks*, onde cada *lock* estivesse associado a uma célula, por exemplo, vários processos poderiam ser executados ao mesmo tempo sobre a estrutura, às custas de uma maior complexidade dos algoritmos. Tal complexidade é oriunda do controle da interferência entre múltiplas operações e do manejo de vários *locks* ao mesmo tempo (o que implica, como veremos

adiante, em muitas dificuldades).

Implementação

Podemos implementar os *locks* de uma maneira simples usando uma primitiva de *hardware* chamada *compare-and-swap* (CAS) [10]. Essa primitiva está presente em diversos processadores, e funciona conforme o algoritmo 20, `CAS()`, que mostramos adiante:

Algoritmo 20 CAS(v, t, c)

1: **if** $v = t$ **then**
2: $\quad v \leftarrow c$
3: $\quad$ **return** *true*
4: **return** *false*

Porém, `CAS()` não é uma rotina normal. Ela tem sido chamada de primitiva já que é normalmente implementada em *hardware*, e é atômica no seguinte sentido: quando opera sobre uma variável qualquer v, seus passos são executados sem intercalar com outras operações do mesmo ou de qualquer outro processo que porventura manipule v. Em outras palavras, em um contexto paralelo, um processo só consegue iniciar uma operação qualquer sobre v se ninguém está a executar a primitiva `CAS()` sobre v.

Desta forma, um *lock* poderia simplesmente ser uma variável inteira que indica estado (1 quando obtido e 0 quando livre), manipulado somente através de primitivas `CAS()`. O *lock* seria iniciado com o valor 0. Em um *lock* l, vamos escrever `obterLock(`l`)` e `liberarLock(`l`)` para obtermos e liberamos l, que seriam feitos, respectivamente, conforme definido nos algoritmos 21 e 22. Novamente, tanto na obtenção quanto na liberação do *lock*, devemos empregar o algoritmo `CAS()` para manipular atomicamente o *lock*. Dessa forma, o *lock* fica sempre com valor 0 ou 1, e nunca com algum outro valor causado por manipulações concorrentes de l.

Algoritmo 21 obterLock(l)

1: **while** CAS(l, 0, 1) = *false* **do**
2: $\quad$ **continue**

Algoritmo 22 liberarLock(l)

1: CAS(l, 1, 0)

O algoritmo anterior para obter *locks* bloqueia o processo requisitante até que o *lock* fosse obtido. Todavia, um processo poderia querer tentar obter um *lock* sem ficar bloqueado. Por exemplo, isso é

necessário para evitar *deadlocks*, tópico discutido mais adiante. Um algoritmo que faça isso deveria retornar imediatamente, indicando se a tentativa de obtenção foi feita com sucesso ou não. Para isso, vamos definir o algoritmo 23, `tentarObterLock(l)`.

Algoritmo 23 tentarObterLock(l)

1: **return** CAS(l, 0, 1)

Privilégios de Acesso

No exemplo acima, as regiões críticas de `modificarConc1()` e `modificarConc2()` são ambas leitura-escrita (*read-write*), ou seja, elas podem ler e modificar os dados associados (no caso, somente x). Porém, nem sempre isso é assim. Muitas vezes, temos várias operações concorrentes onde algumas tenham regiões críticas somente-leitura (*read-only*), que apenas lêem os dados associados, e em outras delas tenham regiões críticas leitura-escrita, que lêem e possivelmente modificam esses dados.

As diversas regiões somente-leitura poderiam ser executadas ao mesmo tempo, desde que não houvesse nenhuma região leitura-escrita sendo executada simultaneamente. Ou seja, as regiões leitura-escrita deveriam ser exclusivas, mas as regiões somente-leitura poderiam ser concorrentes. Essa demanda é bastante usual, e por isso vamos dizer que *locks* são de dois tipos: (1) *locks simples*, que não permitem especificar privilégios especiais de acesso, como os *locks* que temos visto até agora; e *locks compostos*, que permitem especificar, no momento da obtenção, o tipo de acesso que desejamos adquirir (somente-leitura ou leitura-escrita).

Nessa seção, definimos `obterLockLeitura(l)`, `liberarLockLeitura(l)`, `obterLockEscrita(l)` e `liberarLockEscrita(l)` para, respectivamente, obter em modo somente-leitura, liberar em modo somente-leitura, obter em modo leitura-escrita e liberar em modo leitura-escrita um *lock composto* l. Os algoritmos correspondentes são, respectivamente, os de número 24, 25, 26 e 27. Na nossa implementação, os *locks* compostos têm dois campos: *estadoRO* e *estadoRW*. O campo *estadoRO* armazena a quantidade de processos que possuem acesso somente-leitura sobre o recurso associado, e o campo *estadoRW* a quantidade de processos que possuem acesso leitura-escrita ao recurso. O campo *estadoRW* só pode guardar 0 ou 1. Vamos empregar *locks* simples para controlar o acesso aos campos dos *locks* compostos – também precisamos evitar inconsistências no próprio *lock*.

Além disso, os algoritmos 28 e 29, `tentarObterLockLeitura()` e `tentarObterLockEscrita()`, análogos aos descritos anteriormente, tentam obter um *lock* composto sem bloquear o processo que quer obter o *lock*.

Algoritmo 24 obterLockLeitura(l)

Requer: Um *lock* l
Garante: l será obtido em modo somente-leitura
1: obterLock($l.controle$)
2: **while** $l.estadoRW = 1$ **do**
3: liberarLock($l.controle$)
4: obterLock($l.controle$)
5: $l.estadoRO \leftarrow l.estadoRO + 1$
6: liberarLock($l.controle$)

Algoritmo 25 obterLockEscrita(l)

Requer: Um *lock* l
Garante: l será obtido em modo leitura-escrita
1: obterLock($l.controle$)
2: **while** $l.estadoRW = 1$ **or** $l.estadoRO > 0$ **do**
3: liberarLock($l.controle$)
4: obterLock($l.controle$)
5: $l.estadoRW \leftarrow 1$
6: liberarLock($l.controle$)

Algoritmo 26 liberarLockLeitura(l)

Requer: Um *lock* l obtido em modo somente-leitura
Garante: l será liberado
1: obterLock($l.controle$)
2: $l.estadoRO \leftarrow l.estadoRO - 1$
3: liberarLock($l.controle$)

Algoritmo 27 liberarLockEscrita(l)

Requer: Um *lock* l obtido em modo leitura-escrita
Garante: l será liberado
1: obterLock($l.controle$)
2: $l.estadoRW \leftarrow 0$
3: liberarLock($l.controle$)

Algoritmo 28 tentarObterLockLeitura(l)

Requer: Um *lock* l

Garante: Se a função devolve *true*, l estará obtido em modo somente-leitura, caso contrário l não estará obtido em modo somente-leitura

1: obterLock($l.controle$)
2: **if** $l.estadoRW = 1$ **then**
3: liberarLock($l.controle$)
4: **return** *false*
5: $l.estadoRO \leftarrow l.estadoRO + 1$
6: liberarLock($l.controle$)
7: **return** *true*

Algoritmo 29 tentarObterLockEscrita(l)

Requer: Um *lock* l

Garante: Se a função devolve *true*, l estará obtido em modo leitura-escrita, caso contrário l não estará obtido em modo leitura-escrita

1: obterLock($l.controle$)
2: **if** $l.estadoRW = 1$ **or** $l.estadoRO > 0$ **then**
3: liberarLock($l.controle$)
4: **return** *false*
5: $l.estadoRW \leftarrow 1$
6: liberarLock($l.controle$)
7: **return** *true*

Deadlocks

Os *deadlocks* são situações indesejáveis em que um grupo de operações concorrentes fica bloqueado, de forma que o andamento dessas operações dependa do progresso de operações do próprio grupo para continuar. Portanto, o grupo de operações fica bloqueado sem haver andamento de qualquer operação.

Considere um grupo de dados $d_1, \ldots, d_n$, associados respectivamente a *locks* $l_1, \ldots, l_n$. Digamos que uma operação `op1()` precise obter os *locks* l_1, l_2 e l_3 para modificar os dados correspondentes, e que uma outra operação `op2()` precise dos mesmos *locks*. Vamos supor que cada operação esteja associada a um único processo, que os *locks* possam ser obtidos em ordem arbitrária, e que um processo não libere nenhum *lock* até adquirir todos os que precisa. Assim, a seguinte situação poderia ocorrer:

1. O processo A tenta obter l_3 em `op1()`

2. O processo A obtém l_3 em `op1()`

3. O processo A tenta obter l_2 em `op1()`
4. O processo A obtém l_2 em `op1()`
5. O processo B tenta obter l_1 em `op2()`
6. O processo B obtém l_1 em `op2()`
7. O processo A tenta obter l_1 em `op1()`
8. O processo B tenta obter l_2 em `op2()`

Veja que as duas últimas operações de obtenção múltipla dependem do progresso uma da outra para continuar, mas seus processos associados ficam bloqueados eternamente. Se tivéssemos apenas um *lock*, associado a todos os dados, isso nunca iria acontecer. Porém, perderíamos concorrência, já que o acesso a um ou poucos dados bloquearia o acesso aos outros dados. Quando temos grandes conjuntos de dados, como esperamos ter nos *skip graphs*, ter um único *lock* é demasiadamente ineficiente.

Portanto, se quisermos promover uma maior concorrência usando muitos *locks*, precisamos tomar muito cuidado para evitar a situação anterior.

Coffman [8] definiu algumas condições necessárias para que *deadlocks* ocorram. Mais especificamente, basta evitar uma das condições definidas para *garantir* que não teremos *deadlocks*. As condições definidas são:

Exclusão mútua: um recurso não pode ser obtido por mais de um processo;

***Hold and wait*:** um processo pode bloquear à espera de um recurso enquanto já possui um ou mais recursos;

Não-preempção: quando bloqueado, um processo não pode ser desbloqueado;

Grafo de espera circular: consideremos um grafo dinâmico cujos vértices são processos e recursos. No nosso ambiente concorrente, enquanto um processo tenta obter um recurso, existe no nosso grafo uma aresta que vai do processo ao recurso; enquanto um recurso é de posse de um processo, existe uma aresta que vai do recurso ao processo. Se não surgir um circuito nesse grafo, não temos uma situação de *deadlock*.

Precisamos proporcionar exclusão mútua, por isso estamos usando *locks*, logo não podemos evitar a última condição. Nos algoritmos concorrentes a serem apresentados, poderíamos tentar obter os *locks* sempre das células associadas com menores chaves primeiro, tentando evitar a condição do grafo

de espera circular. Porém, isso nem sempre será possível, como veremos adiante. Então, precisaremos eventualmente obter *locks* de uma forma que não use o paradigma *hold-and-wait*. Com isso vamos evitar *deadlocks*, como mostraremos posteriormente.

Voltando aos algoritmos relacionados a *locks*, vamos escrever `obterLocks(`$l_1, \ldots, l_n$`)` para denotar um algoritmo que obtém os *locks* simples $l_1, \ldots, l_n$, na ordem, em modo *hold-and-wait*. Similarmente, para *locks* compostos, definimos `obterLocksLeitura(`$l_1, \ldots, l_n$`)` e `obterLocksEscrita(`$l_1, \ldots, l_n$`)`. Temos também os algoritmos que liberam os *locks*: `liberarLocks(`$l_1, \ldots, l_n$`)` (para *locks* simples), `liberarLocksLeitura(`$l_1, \ldots, l_n$`)` e `liberarLocksEscrita(`$l_1, \ldots, l_n$`)` (essas últimas para *locks* compostos).

Finalmente, definimos o algoritmo 30, `tentarObterLocksEscrita()`, que recebe *locks* $l_1, \ldots, l_n$ e tenta obter todos em modo leitura-escrita, mas não em um modo *hold-and-wait*. Ela devolve *true* se foi possível obter todos os *locks*, em modo leitura-escrita, e devolve *false* em caso contrário. Como dissemos, isso será um dos artifícios usados contra *deadlocks*.

Algoritmo 30 tentarObterLocksEscrita($l_1, \ldots, l_n$)

Requer: Vários *locks* $l_1, \ldots, l_n$

Garante: Se a função devolve *true*, $l_1, \ldots, l_n$ estarão obtidos em modo leitura-escrita, caso contrário $l_1, \ldots, l_n$ não estarão obtidos em modo leitura-escrita

```
1: for i ← 1 to n do
2:   if tentarObterLockEscrita(l_i) = false then
3:     for j ← 1 to i − 1 do
4:       liberarLockEscrita(l_j)
5:     return false
6: return true
```

4.3 Algoritmos

Nesta seção, vamos mostrar algoritmos concorrentes que operam sobre um *skip graph*. Visando uma boa concorrência das operações, cada célula c vai ter um *lock* composto $l_o(c)$, chamado de *lock* de operações de c. Esse *lock* será usado para sincronizar operações somente-leitura e leitura-escrita que modificam os campos da célula. As operações de que estamos falando são coisas como consultar ou modificar o nível da célula, o valor de uma referência, etc. Além dele, cada célula vai ter também um *lock* simples $l_p(c)$, chamado de *lock* de promoção de c, cuja utilidade será mostrada mais adiante.

Na nossa abordagem, seria possível empregar *locks* individuais para os campos *nivel*, *palavra* e para cada referência, aumentando potencialmente a concorrência das operações. Porém, a atribuição de uma nova vizinha poderia mudar o nível de uma célula, e portanto algo simples assim já exigiria utilizar múltiplos *locks*. Desta forma, nossos algoritmos iriam ficar bem mais complicados, então

preferimos a opção mais simples e mais clara. Uma análise informal não evidencia muitos ganhos na abordagem que usa um *lock* para cada campo das células.

4.3.1 Infra-estrutura

Nos nossos algoritmos, qualquer chamada a `atribuirNível()` e `atribuirVizinho()` será feita apenas quando tivermos obtido previamente o *lock* de operações da célula operada c em modo leitura-escrita. A liberação do *lock* será feita pelo chamador, após essas chamadas terminarem. Desta forma, podemos garantir que enquanto atualizamos certos campos de c, não existe outra célula c' que esteja lendo simultaneamente esses campos.

Para facilitar, vamos definir *wrappers* para os algoritmos de obtenção e liberação de *locks*, que passam a receber as células cujos *locks* serão obtidos e liberados:

- `obterLock(`c`)`, `liberarLock(`c`)` e também `obterLockLeitura(`c`)`, `liberarLockLeitura(`c`)`, `obterLockEscrita(`c`)` e `liberarLockEscrita(`c`)` são como os algoritmos análogos se $c \neq \perp$, e em caso contrário eles não fazem nada;
- `tentarObterLock(`c`)`, `tentarObterLockLeitura(`c`)` e `tentarObterLockEscrita(`c`)` são como os algoritmos análogos se $c \neq \perp$, e em caso contrário eles apenas devolvem *true*;
- Da mesma forma, no caso dos algoritmos que recebiam vários *locks* ao mesmo tempo, vamos definir *wrappers* que recebem várias células como parâmetros, obtêm ou liberam *locks* das que são $\neq \perp$, e ignoram os parâmetros $= \perp$. Se um *wrapper* devesse devolver um valor booleano (como `obterLocks(`$c_1, \ldots, c_n$`)`), e todos os parâmetros forem iguais a $\perp$, vamos definir que o valor agora devolvido será *true*.

Esses *wrappers* evitam poluir nossos algoritmos com muitos testes que verificam se uma referência é $\neq \perp$ nas diversas obtenções ou liberações de *locks*.

4.3.2 Busca

O algoritmo concorrente de busca no *skip graph* não é muito diferente do correspondente tradicional, com exceção dos *locks* obtidos e liberados no decorrer dos algoritmos. O algoritmo 31, `buscar(`c`, `j`, `e`)`, em sua versão concorrente, recebe uma célula $c \neq \perp$, cujo *lock* de operações foi obtido em modo somente-leitura, um nível $j \geq 0$ e um elemento $e \in U$, e devolve a célula que vem logo antes de e incluindo e se $c.chave < e$, a célula que vem logo depois de e incluindo e se $c.chave > e$ ou, trivialmente, a célula c, se $c.chave = e$. A célula c tem seu *lock* de operações obtido em modo somente-leitura. Da mesma forma que o algoritmo tradicional, o algoritmo é tipicamente invocado com $j = c.nivel$. Observe que o percurso realizado pelo algoritmo concorrente é o mesmo percurso

realizado pelo algoritmo tradicional, mas sempre obtendo o *lock* de operações da célula que será logo visitada e liberando o *lock* de operações da célula anterior no percurso.

Observe que, ao final da busca, a célula devolvida pode estar em processo de inserção ou de remoção, por causa da possibilidade de inserções ou de remoções executadas concorrentemente.

Algoritmo 31 buscar(c, j, e)

```
1:  ▷ O lock de operações de c foi obtido previamente em modo somente-leitura
2:  if c.chave < e then
3:    c' ← c.prox[j]
4:    obterLockLeitura(c')
5:    if c' ≠ ⊥ and c'.chave ≤ e then
6:      liberarLockLeitura(c)
7:      return buscar(c', j, e)
8:    else
9:      liberarLockLeitura(c')
10:     if j ≥ 1 then
11:       return buscar(c, j − 1, e)
12:     else
13:       return c
14: if c.chave > e then
15:   c' ← c.ante[j]
16:   obterLockLeitura(c')
17:   if c' ≠ ⊥ and c'.chave ≥ e then
18:     liberarLockLeitura(c)
19:     return buscar(c', j, e)
20:   else
21:     liberarLockLeitura(c')
22:     if j ≥ 1 then
23:       return buscar(c, j − 1, e)
24:     else
25:       return c
26: return c ▷ l_o(c) foi obtido em modo somente-leitura, mas não liberado
```

Uma maneira de evitarmos *deadlocks* seria fazer com que *locks* fossem obtidos sem nunca gerar os ciclos descritos na seção 4.2. Para isso, uma possível abordagem seria obter *locks* de células de chaves menores antes de *locks* de células de chaves maiores. Porém, determinados algoritmos dos *skip graphs*, como a própria busca, precisam caminhar em ambas as direções, tornando essa abordagem inviável.

Apesar disso, na busca, os *locks* são sempre obtidos em modo somente-leitura, o que nos permite plena concorrência com diversas operações que também são somente-leitura. Surgem problemas

apenas quando temos obtenções de múltiplos *locks* em modo leitura-escrita. Como veremos adiante, vamos empregar artifícios diferentes para neutralizar a possibilidade de ocorrência de *deadlocks*.

4.3.3 Inserção

O algoritmo 32, `inserir(`a`, `e`)`, permite fazer a inserção concorrente em um *skip graph*. Ele recebe uma célula a (o apresentador) e um elemento $e \in U$. Se $a \neq \perp$, o *lock* de operações de a deve estar obtido em modo somente-leitura. Se o *skip graph* já contém uma célula com chave e, o algoritmo não modifica a estrutura. Caso contrário, o algoritmo tenta obter, em modo leitura-escrita, os *locks* de operações das células de S_ϵ entre as quais a nova célula deveria ser inserida. Se essa tentativa obtiver sucesso, o algoritmo insere na posição correta da estrutura uma nova célula cuja chave é e, atualiza todas as referências e devolve *true*. Caso contrário, a inserção é abortada e o algoritmo devolve *false*. Da mesma forma que antes, se $a = \perp$, o algoritmo 13 cria um novo *skip graph* contendo uma única célula, cuja chave é e. Caso contrário, a vai ser usada para verificar se o elemento e já está presente no *skip graph*, e, se não estiver, para encontrar a posição em S_ϵ onde será inserida a nova célula com chave e.

Observe que `inserir()`, sempre que acessa uma célula, está de posse do seu *lock* de operações em modo somente-leitura, e sempre que altera o conteúdo de uma célula, está de posse do seu *lock* de operações em modo leitura-escrita. Isso é feito de uma maneira que tenta minimizar o período em que um *lock* de operações fica obtido pelo algoritmo.

As inserções nos níveis > 0 são feitas no algoritmo 35, `inserirNiveisSuperiores()`, discutido posteriormente. O algoritmo `inserir()` é o primeiro algoritmo que apresenta operações leitura-escrita no *skip graph*. Ao contrário dos algoritmos mostrados até agora, vamos precisar obter, ao mesmo tempo, vários *locks* em modo leitura-escrita. Como veremos, "algoritmos leitura-escrita" são bem mais complexos que as versões tradicionais correspondentes.

Para controlar a concorrência dentre as operações na estrutura, adicionamos três novos campos em cada célula: *inserindo*, que indica que a célula está em processo de inserção, *removendo*, que indica que a célula está em processo de remoção, e *nivelInserindo*, que indica o último nível em que uma célula em processo de inserção já se inseriu. Assumimos que `alocarCélula()` agora inicia esses campos com *true*, *false* e -1, respectivamente.

Antes da invocação de `inserir(`a`, `e`)`, devemos possuir $l_o(a)$ em modo somente-leitura. No algoritmo, verificamos inicialmente se a célula apresentadora a está sendo inserida ou removida. O estado de a não pode ser modificado enquanto possuímos seu *lock* de operações, logo entre essa verificação em a e a busca partindo de a, seu estado continua o mesmo. A razão pela qual não queremos utilizar células sendo inseridas ou removidas como ponto de partida de uma busca é porque

isso poderia comprometer a eficiência da inserção, já que as referências de tal célula podem estar ainda sendo construídas ou sendo anuladas.

Na inclusão da nova célula em um nível j, vamos tentar obter os *locks* de operações das células envolvidas em modo leitura-escrita. Essas células são a própria célula sendo incluída e as suas futuras vizinhas na lista ligada de nível j correspondente. Antes, não precisávamos que nenhuma mudança realizada pela inserção fosse atômica, pois nenhuma outra operação era feita ao mesmo tempo. Agora, temos várias outras operações sendo feitas ao mesmo tempo, e uma das precauções que vamos tomar é fazer com que as inclusões de células nas listas ligadas sejam atômicas.

Porém, em todos os nossos algoritmos, nunca vamos obter *locks* de operações em modo leitura-escrita diretamente. Primeiro, vamos obter esses *locks* em modo somente-leitura, e então usar `tentarPromoverLocks()` (veja algoritmo 33) para convertê-los em *locks* obtidos em modo leitura-escrita. Em tal algoritmo, os *locks* obtidos em modo somente-leitura são liberados, e então reobtidos em modo leitura-escrita, sem que as células associadas sejam modificadas durante o processo. A maneira específica pela qual isso será feito é discutida na seção 4.3.4.

A inclusão da célula nos níveis superiores é discutida na seção 4.3.5.

4.3.4 Promoção de *Locks*

Quando temos *locks* obtidos em modo somente-leitura, mas queremos obter esses *locks* em modo leitura-escrita, não há outra alternativa senão liberar os *locks* obtidos em modo somente-leitura e obtê-los novamente em modo leitura-escrita. O problema é que entre a liberação e a obtenção dos *locks*, a célula associada poderia ser modificada por outros processos.

Para fazer essa "promoção de *locks*", digamos em $l_o(c)$, vamos usar a seguinte técnica: (1) vamos tentar obter o *lock* de promoção de c, $l_p(c)$; (2) se obtivermos com sucesso o *lock* de promoção de c, podemos liberar $l_o(c)$ do modo somente-leitura e então obter $l_o(c)$ em modo leitura-escrita; (3) se não obtivermos com sucesso o *lock* de promoção de c, apenas liberamos $l_o(c)$.

Dessa forma, se qualquer obtenção de *lock* em modo leitura-escrita for feita através da maneira anterior, e qualquer escrita em uma célula for feita apenas quando obtido o *lock* correspondente, nenhum processo poderia escrever em uma célula cujo *lock* estivesse envolvido em um processo de promoção. A obtenção do *lock* de promoção é feita sempre usando `tentarObterLock()`, já que não queremos esperar pelo *lock* de promoção: se outro processo obtiver esse *lock*, a célula associada poderá ser modificada.

Para promover um único *lock*, vamos usar o algoritmo 33, `tentarPromoverLock(`c`)`. Este algoritmo recebe uma célula c e devolve *true* se $c = \perp$ ou se $c \neq \perp$ e foi também possível obter o *lock*

Algoritmo 32 inserir(a, e)

1: ▷ Se $a \neq \perp$, o *lock* de operações de a foi obtido previamente em modo somente-leitura
2: $c \leftarrow$ alocarCélula(e, -1)
3: **if** $a = \perp$ **then**
4:    atribuirNível(0)
5:    $c.inserindo$ = false
6:    **return** *true* ▷ Primeiro elemento: não há mais nada a fazer
7: **if** $a.inserindo$ **or** $a.removendo$ **then**
8:    liberarCelula(c)
9:    liberarLockLeitura(a)
10:    **return** *false* ▷ Não usamos células sendo inseridas ou removidas como apresentadores
11: $c' \leftarrow$ buscar(a, $a.nivel$, e) ▷ Ao final, $l_o(c')$ foi obtido em modo somente-leitura, mas não liberado
12: **if** $c'.chave = e$ **then**
13:    liberarCelula(c)
14:    liberarLockLeitura(c') ▷ O elemento já está presente
15:    **return** *false*
16: **else if** $c'.chave < e$ **then**
17:    $c_e \leftarrow c'$
18:    $c_d \leftarrow c_e.prox[0]$
19:    obterLockLeitura(c_d)
20: **else if** $c'.chave > e$ **then**
21:    $c_d \leftarrow c'$
22:    $c_e \leftarrow c_d.ante[0]$
23:    obterLockLeitura(c_e)
24: obterLockLeitura(c)
25: **if not** tentarPromoverLocks(c_e, c, c_d) **then**
26:    liberarLocksLeitura(c_e, c, c_d) ▷ As células estão sendo usadas
27:    **return** *false*
28: atribuirNível(c, 0)
29: atribuirVizinho(c, c_e, E, 0)
30: atribuirVizinho(c, c_d, D, 0)
31: **if** $c_e \neq \perp$ **then**
32:    atribuirVizinho(c_e, c, D, 0)
33: **if** $c_d \neq \perp$ **then**
34:    atribuirVizinho(c_d, c, E, 0)
35: $c.nivelInserindo \leftarrow 0$
36: liberarLocksEscrita(c_e, c, c_d)
37: inserirNiveisSuperiores(c)

de operações de c em modo leitura-escrita, e *false* em caso contrário. Para promover vários *locks*, vamos usar o algoritmo 34, `tentarPromoverLocks(`$c_1, \ldots, c_n$`)`. Este algoritmo recebe um conjunto de células $c_1, \ldots, c_n$ e devolve *true* se foi possível obter o *lock* de operações de todas as células $c_i \neq \perp$, para $1 \leq i \leq n$, em modo leitura-escrita, e *false* em caso contrário. Todas as tentativas de obtenção de *locks* são feitas usando `tentarObterLockEscrita()`, ou seja, não são bloqueantes.

Algoritmo 33 tentarPromoverLock(c)

```
1: if c = ⊥ then
2:   return true
3: if tentarObterLock(l_p(c)) then
4:   if tentarObterLockEscrita(l_o(c)) then
5:     liberarLock(l_p(c))
6:     return true
7:   liberarLock(l_p(c))
8: return false
```

Algoritmo 34 tentarPromoverLocks($c_1, \ldots, c_n$)

```
1: for i ← 1 to n do
2:   if c_i ≠ ⊥ then
3:     if tentarObterLock(l_p(c_i)) = false then
4:       for j ← 1 to i − 1 do
5:         liberarLock(l_p(c_i))
6:       return false
7: for i ← 1 to n do
8:   if c_i = ⊥ then
9:     if tentarObterLockEscrita(l_o(c_i)) = false then
10:      for j ← 1 to i − 1 do
11:        liberarLockEscrita(l_o(c_j))
12:      for j ← 1 to n do
13:        liberarLock(l_p(c_j))
14:      return false
15: for j ← 1 to n do
16:   if c_j ≠ ⊥ then
17:     liberarLock(l_p(c_i))
18: return true
```

4.3.5 Escolha das vizinhas compatíveis

Na inserção da célula nos níveis > 0, temos muitas diferenças com relação ao comportamento do algoritmo tradicional. Não podemos encontrar as primeiras células compatíveis de cada lado e simplesmente torná-las vizinhas de c. O algoritmo 35, `inserirNiveisSuperiores(`c`)`, recebe uma

célula c que foi inserida somente em S_ϵ, e faz a inserção nas outras listas ligadas, de níveis > 0. O algoritmo 37, `determinarVizinhosCompatAcima()`, é usado para determinar, em cada nível, o par de células vizinhas em um nível acima entre as quais a nova célula será inserida. Este último algoritmo, assim como o bloco da linha 9 serão explicados posteriormente.

Algoritmo 35 inserirNiveisSuperiores(c)

1: $j \leftarrow 0$
2: **while true do**
3: obterLockLeitura(c)
4: $(c_e, c_d) \leftarrow$ determinarVizinhosCompatAcima(c, j)
5: ▷ Ao final, as células $\neq \perp$ terão seus *locks* obtidos em modo somente-leitura
6: **if not** tentarPromoverLocks(c_e, c, c_d) **then**
7: liberarLocksLeitura(c_e, c, c_d)
8: **continue**
9: **if** $c_e = \perp$ **and** $c_d = \perp$ **then**
10: $c_e \leftarrow$ encontrarVizinhoCompatAcima($c.ante[j]$, c, j, false)
11: **if** $c_e = \perp$ **then**
12: **break**
13: liberarLocksEscrita(c_e, c, c_d)
14: **continue**
15: atribuirVizinho(c, c_e, E, $j + 1$)
16: atribuirVizinho(c, c_d, D, $j + 1$)
17: **if** $c_e \neq \perp$ **then**
18: atribuirVizinho(c_e, c, D, $j + 1$)
19: **if** $c_d \neq \perp$ **then**
20: atribuirVizinho(c_d, c, E, $j + 1$)
21: $c.nivelInserindo \leftarrow j$
22: liberarLocksEscrita(c_e, c, c_d)
23: $j \leftarrow j + 1$
24: $c.nivelInserindo \leftarrow j$
25: $c.inserindo \leftarrow false$
26: liberarLocksEscrita(c_e, c, c_d)
27: **return** *true*

Primeiramente, vamos dizer que uma célula c já está inserida no nível i se:

1. $c.inserindo =$ false e $c.nivel \geq i$;
2. $c.inserindo =$ true e $c.nivelInserindo \geq i$.

Note que isso implica que as linhas 15 a 22 do algoritmo 35 já foram executadas no nível i, e portanto implica que a célula em questão já possui vizinhas de nível i.

O algoritmo 36, `estáInserida(c, j)`, recebe uma célula c e um nível $j \geq 0$, e verifica se c já está inserida no nível j, conforme antes definido. Ela será usada por `determinarVizinhosCompatAcima()`, que determina o par de células vizinhas inseridas no nível superior entre as quais nos inserimos.

Algoritmo 36 estáInserida(c, j)

1: **if** $c \neq \perp$ **and** ($c.inserindo$ = false **or** $c.nivelInserindo \geq j$) **then**
2:     **return** *true*
3: **return** *false*

Ao inserir uma nova célula c em uma lista ligada de nível $i + 1$, continuamos percorrendo as listas ligadas de nível i e verificando a compatibilidade entre as palavras aleatórias, mas agora vamos encontrar uma única vizinha, em qualquer direção, que além de ser compatível no nível $i + 1$, já esteja inserida no nível $i + 1$. Chamemos essa vizinha de c_v. A célula c será inserida entre c_v e a vizinha de c_v na direção oposta à que foi encontrada c_v, no nível $i + 1$. Esse cenário é mostrado nas figuras 4.1 e 4.2.

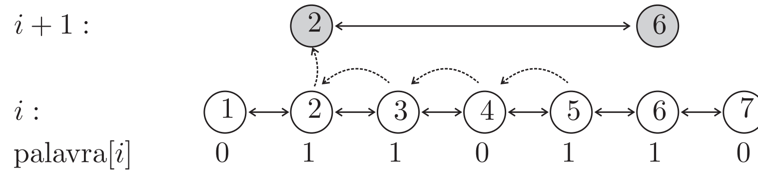

Figura 4.1: *Busca por um vizinho compatível com 5 em uma única direção (esquerda), presente no nível superior. Na figura, 3 é compatível com 5, mas ela é ignorada por ainda não estar inserida no nível $i + 1$.*

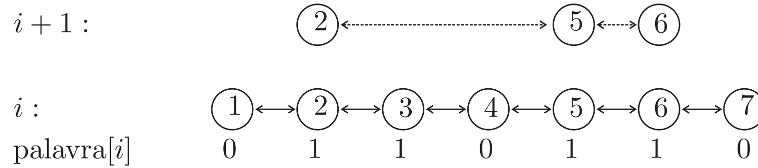

Figura 4.2: *Inserção propriamente dita entre o vizinho compatível previamente encontrado e seu vizinho no nível superior.*

O motivo pelo qual vamos empregar essa tática é que, caso usássemos o algoritmo tradicional com inserções concorrentes, poderia ocorrer o cenário da figura 4.3. Na figura, as células destacadas são células que estão inseridas no nível correspondente.

Na figura 4.3, não existe sempre uma reciprocidade das referências, ou seja, é possível que em determinados instantes, uma célula c_1 seja vizinha à esquerda de uma célula c_2 mas c_2 não seja vizinha à direita de c_1. São formadas bifurcações em listas ligadas, e os algoritmos que caminham através dessas listas ligadas, como `encontrarVizinhoCompatAcima()`, deveriam então lidar com tal

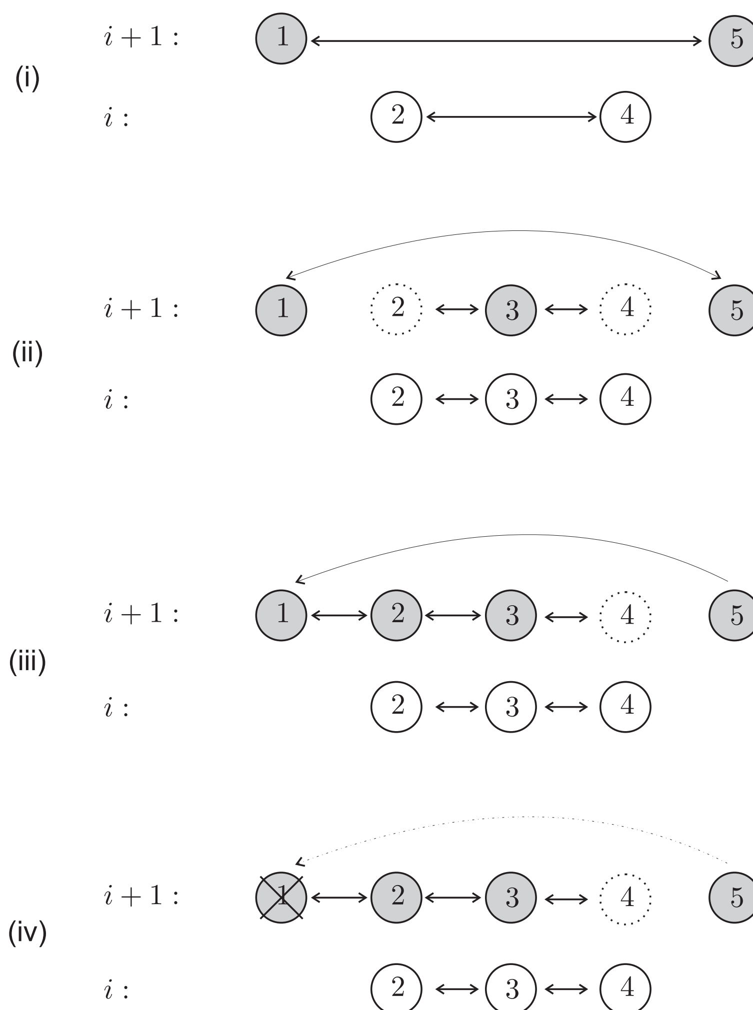

Figura 4.3: *(i) 1 e 5 estão inseridas no nível $i+1$, 2 e 4 estão inseridas no nível i, mas não estão inseridas no nível $i+1$; (ii) 3 se insere entre 2 e 4 nos níveis i e $i+1$, fazendo com que uma célula inserida no nível $i+1$ tenha vizinhos ainda não inseridos no nível $i+1$; (iii) 2 se insere no nível $i+1$, fazendo com que listas ligadas fiquem bifurcadas, o que já dificulta bastante os algoritmos que as percorrem; (iv) se 1 for removida, e a referência de 5 para 1 seria tornada inválida.*

situação (o que é obviamente complicado, considerando que temos inserções e remoções concorrentes em estruturas bifurcadas). Na figura, vimos ainda o que poderia ocorrer em uma remoção da célula 1, e logo seria também necessário encontrar as células que têm 1 como vizinha, atualizando as referências que apontam para a célula 1. Isso é uma operação complexa em uma estrutura com as bifurcações discutidas: nesse contexto, nossos algoritmos que caminham entre listas ligadas estão comprometidos, como vimos acima. Além disso, mesmo quando achássemos uma célula com uma referência para 1, seria complicado determinar um novo valor ao campo referência dessa célula. Portanto, visando

algoritmos mais claros e simples, é melhor evitar a ocorrência de bifurcações.

As bifurcações mencionadas ocorrem porque novas células atribuem como vizinhas, em um certo nível $i+1$, outras células que ainda não foram inseridas no nível $i+1$. Para evitá-las, é preciso que as vizinhas recebidas já estejam inseridas no nível $i+1$ e que elas sejam realmente as vizinhas em tal condição mais próximas no nível $i+1$, conforme o algoritmo 37, `determinarVizinhosCompatAcima()`.

O algoritmo 37, `determinarVizinhosCompatAcima(`c`, `j`)`, recebe uma célula $c \neq \perp$ e um nível $j \geq 0$, e devolve um par de células $(c_e,\ c_d)$. Se $c_e \neq \perp$, c_e está inserida no nível $i+1$, é a primeira célula compatível com c que vem antes de c na lista ligada de nível $i+1$, tem c_d como vizinha à direita de nível $i+1$ e tem seu *lock* de operações obtido em modo somente-leitura; se $c_d \neq \perp$, c_d está inserida no nível $i+1$, é a primeira célula compatível com c que vem depois de c na lista ligada de nível $i+1$, tem c_e como vizinha à esquerda de nível $i+1$ e tem seu *lock* de operações obtido em modo somente-leitura. Se não houverem células que satisfaçam tais condições, o algoritmo devolve $(\perp, \perp)$.

Algoritmo 37 determinarVizinhosCompatAcima(c, j)

1: obterLockLeitura($c.ante[j]$)
2: $c_e \leftarrow$ encontrarVizinhoCompatAcima($c.ante[j]$, c, E, j, true)
3: ▷ Se $c_e \neq \perp$, $l_o(c_e)$ terá sido obtido em modo somente-leitura, mas não liberado
4: obterLockLeitura($c.prox[j]$)
5: $c_d \leftarrow$ encontrarVizinhoCompatAcima($c.prox[j]$, c, D, j, true)
6: ▷ Se $c_d \neq \perp$, $l_o(c_d)$ terá sido obtido em modo somente-leitura, mas não liberado
7: **if** estáInserida(c_e, $j+1$) **then**
8:    liberarLockLeitura(c_d)
9:    $c_d \leftarrow c_e.prox[j+1]$
10:    obterLockLeitura(c_d)
11:    **return** (c_e, c_d)
12: **else if** estáInserida(c_d, $j+1$) **then**
13:    liberarLockLeitura(c_e)
14:    $c_e \leftarrow c_d.ante[j+1]$
15:    obterLockLeitura(c_e)
16:    **return** (c_e, c_d)
17: **return** $(\perp, \perp)$

O algoritmo 38, `encontrarVizinhoCompatAcima(`c'`, `c''`, `d`, `j`, `i`)`, recebe uma célula c' (eventualmente $\perp$), uma célula $c'' \neq \perp$, uma direção $d \in \{D, E\}$, um nível $j \geq 0$ e um indicador booleano i, e devolve a primeira célula compatível com c' que vem antes de $c'.chave$, se $d = E$, ou que vem depois de $c'.chave$, se $d = D$, dentre quaisquer células x que satisfaçam as seguintes restrições:

1. x está inserida no nível j;

2. se $i = true$, x está inserida no nível $j + 1$.

Se uma célula for encontrada, conforme descrita anteriormente, a primeira restrição implica que tal célula pertence à lista ligada de nível j à qual também pertence c''. Se tal célula não existir, o algoritmo devolve $\perp$. Na invocação do algoritmo, se $c' \neq \perp$ e $c'' \neq \perp$, c' e c'' devem ser compatíveis no nível j. Pelo mesmo motivo falado no capítulo anterior, só precisamos comparar o $(j + 1)$-ésimo símbolo das palavras aleatórias de c' e c'' para verificar a compatibilidade entre essas células no nível $j + 1$.

Quando é encontrada uma célula compatível, que também satisfaz as restrições anteriores, c_f, devemos procurar, na direção oposta a d e no nível $j + 1$, por células presentes entre a célula encontrada c_f e a célula original c''. Isso é necessário já que uma outra célula compatível, que também satisfaz as restrições anteriores, pode ter se inserido entre c_f e c'' no decorrer da execução de `encontrarVizinhoCompatAcima()`. Tal percurso adicional é feito por `encontrarMelhorVizinho()` (mostrada adiante).

Algoritmo 38 encontrarVizinhoCompatAcima(c', c'', d, j, i)

1: **if** $c' = \perp$ **then**
2:   **return** $\perp$
3: ▷ O *lock* de operações de c' foi obtido previamente em modo somente-leitura
4: **if** $d = E$ **then**
5:   **if** $c'.palavra[j] = c''.palavra[j]$ **and** (i = false **or** estáInserida(c', $j + 1$)) **then**
6:     **return** encontrarMelhorVizinho(c', c'', D, $j + 1$)
7:   **else**
8:     obterLockLeitura($c'.ante[j]$)
9:     liberarLockLeitura(c)
10:     **return** encontrarVizinhoCompatAcima($c'.ante[j]$, c'', d, j, i)
11: **else if** $d = D$ **then**
12:   **if** $c'.palavra[j] = c''.palavra[j]$ **and** (i = false **or** estáInserida(c', $j + 1$)) **then**
13:     **return** encontrarMelhorVizinho(c', c'', E, $j + 1$)
14:   **else**
15:     obterLockLeitura($c'.prox[j]$)
16:     liberarLockLeitura(c)
17:     **return** encontrarVizinhoCompatAcima($c'.prox[j]$, c'', d, j, i)

O algoritmo 39, `encontrarMelhorVizinho(`c`, `c'`, `d`, `j`)`, recebe uma célula c, uma célula c', uma direção $d \in \{D, E\}$ e um nível $j \geq 0$. Se $c \neq \perp$, o *lock* de operações de c deve estar obtido em modo somente-leitura. O algoritmo devolve a última célula que vem antes de $c'.chave$ (se $d = D$) ou a primeira célula que vem depois de $c'.chave$ (se $d = E$), na lista ligada de nível j a que pertence c. A maneira pela qual esse algoritmo obtém e libera os *locks* das células é igual à feita nas buscas.

Algoritmo 39 encontrarMelhorVizinho(c, c', d, j)

1: **if** $c = \bot$ **then**
2:    **return** $\bot$
3: $\triangleright$ O *lock* de operações de c foi obtido previamente em modo somente-leitura
4: **if** $d = E$ **then**
5:    $c_v \leftarrow c.ante[j]$
6:    obterLockLeitura(c_v)
7:    **if** $c_v = \bot$ **or** $c_v.chave > c'.chave$ **then**
8:       liberarLockLeitura(c)
9:       **return** encontrarMelhorVizinho(c_v, c', d, j)
10: **else if** $d = D$ **then**
11:    $c_v \leftarrow c.prox[j]$
12:    obterLockLeitura(c_v)
13:    **if** $c_v = \bot$ **or** $c_v.chave < c'.chave$ **then**
14:       liberarLockLeitura(c)
15:       **return** encontrarMelhorVizinho(c_v, c', d, j)
16: liberarLockLeitura(c_v)
17: **return** c

Ainda temos alguns problemas. Quando `determinarVizinhosCompatAcima()` devolve $(\bot, \bot)$, não se pode mais dizer que fomos a primeira célula a se inserir no nível $i + 1$, e terminar a inserção em `inserirNiveisSuperiores()`. Agora, com inserções concorrentes, é possível que várias células compatíveis em um nível $i + 1$ se insiram em momentos próximos em uma mesma lista ligada de nível j, e que várias delas executem `determinarVizinhosCompatAcima()`, obtendo $(\bot, \bot)$. Veja a figura 4.4.

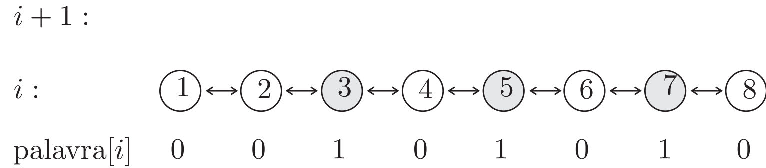

Figura 4.4: *Se as células destacadas, não inseridas no nível $i + 1$, forem formar uma lista ligada de nível $i + 1$, sendo todas compatíveis nesse nível, várias delas poderiam encontrar $(\bot, \bot)$ como vizinhas compatíveis no nível $i + 1$, pois não existem outras células compatíveis já inseridas nesse nível.*

Portanto, se uma nova célula c encontrar $(\bot, \bot)$ como vizinhas superiores no nível $i + 1$, ela vai tentar promover somente $l_o(c)$ na linha 6 do algoritmo 35, `inserirNiveisSuperiores()`, e em seguida verificar se ela é a célula mais à esquerda no nível i (linha 10 do algoritmo 35). Para isso, ela tenta achar uma vizinha compatível à esquerda, essa estando ou não inserida no nível $i + 1$. Se uma vizinha compatível for encontrada, a inclusão no nível em questão é reiniciada (linha 8 do algoritmo 35). A única célula que vai ter sucesso é a célula com menor chave dentre as que

encontraram $(\perp, \perp)$. Esta célula vai ser a primeira incluída no nível $i+1$, com vizinhas ambas nulas. As outras células, quando repetirem o laço da linha 2 do algoritmo 35, já irão encontrar pelo menos uma célula inserida no nível em questão.

4.3.6 Remoção

O algoritmo 40, `remover(`a`, `e`)`, permite fazer a remoção concorrente em um *skip graph*. Ele recebe uma célula $a \neq \perp$ (o apresentador) e um elemento $e \in U$. O *lock* de operações de a deve estar obtido em modo somente-leitura. Se o *skip graph* não contém nenhuma célula cuja chave é e, o algoritmo não modifica a estrutura. Caso contrário, o algoritmo remove da estrutura a célula cuja chave é e e atualiza todas as referências. Da mesma forma que antes, a vai ser usada para verificar se o elemento e já está presente no *skip graph*, e, se estiver, para encontrar a célula com chave e que vai ser removida do *skip graph*. Note que esse procedimento exclui totalmente a *skip list* da célula com chave e do *skip graph*.

No algoritmo, procedemos inicialmente como no algoritmo de inserção, verificando se a está sendo inserido ou removido. Na linha 4, se encontramos a célula c a ser removida, também verificamos se c está sendo inserida ou removida. Os motivos são os seguintes: (1) não permitimos remover células ainda sendo inseridas e (2) como várias chamadas concorrentes podem tentar remover c, a célula já pode estar em processo de remoção, e logo não temos que fazer mais nada.

A abordagem empregada na remoção de células é conhecida como "abordagem preguiçosa": primeiro marcamos c como uma célula em processo de remoção, alterando o campo $c.removendo$ para *true*, e somente depois começamos a remover efetivamente c da estrutura. Nas buscas, opcionalmente, podemos ignorar as células em processo de remoção de maneira eficiente, com base no valor do campo *removendo*. Inclusive, podemos ignorar também células em processo de inserção, definindo um algoritmo que busca entre células do *skip graph* que não estão sendo inseridas ou removidas.

Visando facilitar a remoção efetiva de c, o último símbolo de $c.palavra$ é mudado para -1 na linha 14, evitando que c fique aumentando e diminuindo indefinidamente de nível no decorrer da remoção efetiva, por conta de inserções concorrentes. Como nenhuma célula sendo inserida tem tal símbolo em sua palavra aleatória, e somente essas células verificam compatibilidade de vizinhos, isso realmente acontece. Note que também precisamos modificar `atribuirVizinho()`, de forma que, a partir do momento em que uma célula c entre em processo de remoção efetiva, qualquer mudança nas vizinhas de c sempre mantenha o último caractere de $c.palavra$ com o valor -1. Isso é mostrado no algoritmo 41.

Algoritmo 40 remover(a, e)

1: ▷ O *lock* de operações de a foi obtido previamente em modo somente-leitura
2: **if** $a.inserindo$ **or** $a.removendo$ **then**
3:   **return** *false* ▷ A apresentadora não será usada
4: $c \leftarrow$ buscar(a, $a.nivel$, e)
5: **if** $c.chave \neq e$ **then**
6:   liberarLockLeitura(c) ▷ O elemento não está presente
7:   **return** *false*
8: **if** $c.inserindo$ **or** $c.removendo$ **then**
9:   liberarLockLeitura(c)
10:   **return** *false* ▷ A inserção ainda não foi terminada ou a remoção já foi começada
11: **if not** tentarPromoverLock(c) **then**
12:   liberarLockLeitura(c)
13:   **return** *false* ▷ A célula está sendo usada
14: $c.palavra[c.nivel - 1] \leftarrow -1$
15: $c.removendo \leftarrow$ true
16: liberarLockEscrita(c)
17: **while true do**
18:   obterLockLeitura(c)
19:   **if** $(j \leftarrow c.nivel - 1) < 0$ **then**
20:     **break**
21:   $c_e \leftarrow c.ante[j]$
22:   $c_d \leftarrow c.prox[j]$
23:   obterLockLeitura(c_e)
24:   obterLockLeitura(c_d)
25:   **if not** tentarPromoverLocks(c_e, c, c_d) **then**
26:     liberarLocksLeitura(c_e, c, c_d)
27:     **continue**
28:   atribuirVizinho(c, $\perp$, D, j) ▷ Como $c.removendo =$ true, $c.palavra[c.nivel - 1]$ continua $= -1$
29:   atribuirVizinho(c, $\perp$, E, j) ▷ Como $c.removendo =$ true, $c.palavra[c.nivel - 1]$ continua $= -1$
30:   **if** $c_e \neq \perp$ **then**
31:     atribuirVizinho(c_e, c_d, D, j)
32:   **if** $c_d \neq \perp$ **then**
33:     atribuirVizinho(c_d, c_e, E, j)
34:   liberarLocksEscrita(c_e, c, c_d)
35: liberarCélula(c)
36: **return** *true*

Algoritmo 41 atribuirVizinho(c, c', d, j)

1: ▷ O *lock* de operações de c foi obtido previamente em modo leitura-escrita
2: **if** $d = D$ **then**
3: $c.prox[j] \leftarrow c'$
4: **else**
5: $c.ante[j] \leftarrow c'$
6: **if** $c.ante[c.nivel] \neq \perp$ **or** $c.prox[c.nivel] \neq \perp$ **then**
7: atribuirNível(c, $c.nivel + 1$)
8: **if** $c.nivel > 0$ **and** $c.ante[c.nivel - 1] = \perp$ **and** $c.prox[c.nivel - 1] = \perp$ **then**
9: atribuirNível(c, $c.nivel - 1$)
10: **if** $c.removendo$ **then**
11: $c.palavra[c.nivel - 1] \leftarrow -1$

4.4 Análise

Nessa seção, vamos fazer uma análise dos algoritmos concorrentes do *skip graph*. O tempo total que um processo consome bloqueado à espera de *locks* é chamado de *tempo de contenção*, e será provisoriamente desconsiderado, apenas nessa seção. Desta forma, vamos supor que qualquer chamada a `obterLocks()` gaste tempo $O(1)$. Em chamadas a `tentarObterLock()` ou `liberarLock()`, o tempo gasto é $O(1)$. Além disso, em chamadas a `tentarPromoverLocks()`, o tempo gasto é $O(1)$, pois tentamos promover no máximo $O(1)$ *locks* em qualquer invocação desta função nos nossos algoritmos (e ela gasta tempo claramente proporcional ao número de *locks* passados como parâmetros – veja algoritmo 34).

Como estamos tratando de algo dinâmico, cuja estrutura e tamanho variam com o tempo, vamos considerar um *skip graph* entre um instante inicial t_1 e um instante final t_2 arbitrários, mas antes estabelecendo algumas definições. Vamos dizer que uma célula está *estável* em um *skip graph* em um instante t se a execução do algoritmo de inserção para essa célula já terminou em um instante $t_i < t$ e a execução do algoritmo de remoção para essa célula não foi ainda iniciado em um instante $t_r < t$ no *skip graph*. Uma célula está *sendo inserida* no *skip graph* no instante t se o algoritmo `inserir()` está em processo de execução no instante t, e uma célula está *sendo removida* do *skip graph* no instante t se o algoritmo `remover()` está em processo de execução no instante t. O número de células estáveis em um *skip graph* em um instante t será denotado por $s(t)$. O número de células estáveis em uma lista ligada S_w em um instante t será denotado por $s_w(t)$. Desta forma, o número de células estáveis no instante t_1 é $s_\epsilon(t_1) = s(t_1)$ e no instante t_2 é $s_\epsilon(t_2) = s(t_2)$. Analogamente, definimos $i(t)$, $i_w(t)$ e $r(t)$, $r_w(t)$ para o número células sendo inseridas no *skip graph* e sendo removidas do *skip graph*, respectivamente. O total de células do *skip graph* em um instante t será denotado por $n(t)$, e $n(t) = s(t) + i(t) + r(t)$. O total de células em uma lista ligada S_w será denotado por

$n_w(t) = s_w(t) + i_w(t) + r_w(t)$. Quando não especificarmos explicitamente o tipo das células de que estamos falando, vamos supor que estamos tratando do total de células do *skip graph*. O tamanho de um *skip graph* será medido pelo número total de células presentes nele.

O objetivo desta seção é medir o tempo gasto nos algoritmos `inserir()`, `remover()` e `buscar()`, em termos do tamanho do *skip graph*. Já que é variável o tamanho da estrutura, vamos dizer que n é o maior número de células presentes no *skip graph* entre os instantes que estamos considerando: t_1 e t_2. Ou seja, $n = \max\{n(t) : t_1 \leq t \leq t_2\}$. Essa será a variável usada para expressar o tempo gasto nos algoritmos.

Assumimos que qualquer célula tenha chave no intervalo $\{1, \ldots, |U|\}$. Tomemos uma lista ligada qualquer S_w, cujas células sejam $c_1, \ldots, c_n$, onde $c_1.chave < c_2.chave < \cdots < c_{n-1}.chave < c_n.chave$ em um instante t. Vamos definir como *vãos de S_w no instante t* os intervalos $[1, c_1.chave)$, $(c_1.chave, c_2.chave), \ldots, (c_{n-1}.chave, c_n.chave)$, $(c_n.chave, |U|]$. Um *vão de um instante t* é simplesmente um vão de uma lista ligada qualquer no instante t. Além disso, para $1 \leq i \leq j \leq x$, vamos dizer que $c_i, \ldots, c_j$ são células consecutivas de S_w no instante t. Células consecutivas de um instante t são simplesmente células consecutivas de qualquer lista ligada no instante t.

Vamos dizer que um vão precede $i \in \{1, \ldots, |U|\}$ se o segundo componente do intervalo do vão é i, e que um vão sucede $i \in \{1, \ldots, |U|\}$ se o primeiro componente do intervalo do vão é i. Além disso, vamos dizer que vãos $g_1, \ldots, g_x$ são consecutivos se o segundo componente de um vão g_y é igual ao primeiro componente de g_{y+1}, para todo $y \in \{1, \ldots, x-1\}$.

Consideremos portanto um *skip graph* entre instantes t_1 e t_2, que contenha somente células estáveis no instante t_1. Em outras palavras, $n(t_1) = s(t_1)$ e $i(t_1) = r(t_1) = 0$. Vamos chamar as células do instante t_1 de *células originais*. Vamos assumir que, entre t_1 e t_2, sejam feitas no máximo $n(t_1)$ inserções e no máximo $n(t_1)/2$ remoções. Desta forma, entre t_1 e t_2, o *skip graph* terá seu tamanho no máximo duplicado ou reduzido à metade ($n(t_1)/2 \leq n(t_2) \leq 2n(t_1)$).

Para analisar esse cenário, vamos adotar as seguintes hipóteses:

1. `tentarPromoverLocks()` é invocada somente $O(1)$ vezes em cada nível da inserção ou da remoção. Adiante, na seção 5.1, vamos tentar verificar tal hipótese por meio de testes práticos.

2. Em qualquer instante t, o valor da chave correspondente a qualquer inserção é escolhido uniformemente no conjunto $\{1, \ldots, |U|\}$, e o valor da chave correspondente a qualquer remoção é escolhido uniformemente no conjunto das chaves das células presentes na estrutura. Adiante, na seção 5.2.1, verificaremos que tal hipótese não é muito forte, ainda tendo sentido prático.

Com inserções e remoções concorrentes, precisamos verificar se a busca lateral por vizinhos com-

patíveis, feita no processo de inserção de células, ainda gasta tempo $O(\lg n)$. No caso tradicional, é fácil limitar a quantidade de células visitadas lateralmente, já que ela depende somente dos símbolos das palavras aleatórias (veja página 29). No caso concorrente, conforme mostramos, algumas células compatíveis poderiam ser ignoradas, por não estarem presentes no nível acima ao em que está sendo feita a verificação da compatibilidade (isso ocorre quando tais células estão em processo de inserção – veja página 55). Uma célula original não poderia ser ignorada, já que ela não está mais em processo de inserção. Portanto, na nossa análise, vamos precisar conhecer o tamanho esperado de uma seqüência qualquer de células não-originais consecutivas no *skip graph*, em qualquer instante que vai de t_1 até t_2. Para isso, vamos calcular primeiro o número esperado de células não-originais inseridas em um vão do instante t_1 de uma lista ligada qualquer do *skip graph*.

Lema 1. *Entre t_1 e t_2, o número esperado de células não-originais consecutivas inseridas em um vão do instante t_1 de uma lista ligada qualquer é $O(1)$.*

Demonstração. Seja c uma célula não-original de uma lista ligada qualquer do *skip graph*. Seja I o número de células não-originais que sucedem c, incluindo c, na lista ligada em questão, em um instante qualquer entre t_1 e t_2. Conforme assumimos anteriormente, são realizadas no máximo $n(t_1)$ inserções e $n(t_1)/2$ remoções. Então, a probabilidade de que uma célula qualquer, em um instante qualquer entre t_1 e t_2 seja original é $p_o \geq 1/3$, e a probabilidade de que uma célula qualquer seja não-original é $p_n \leq 1/3$. Desta forma, $\Pr[I = x] \leq (p_n)^x \leq (1/3)^x$ e

$$
\begin{aligned}
E[I] &= \sum_{x=0}^{\infty} x \Pr[I = x] \\
&\leq \sum_{x=0}^{\infty} x \left(\frac{1}{3}\right)^x \\
&= \frac{1/3}{(1 - 1/3)^2} \\
&= O(1),
\end{aligned}
$$

já que, se $|a| < 1$, $\sum_{x=0}^{\infty} x a^x = \frac{a}{(1-a)^2}$. □

Em uma lista ligada S_w qualquer, é possível que vãos de S_w no instante t_1 sejam concatenados, formando vãos maiores entre t_1 e t_2. Isso ocorre caso células originais $c_1, \ldots, c_x$, consecutivas na lista ligada S_w no instante t_1, sejam removidas entre t_1 e t_2. Com efeito, os vãos de S_w no instante t_1 que são consecutivos, e precedem e sucedem $c_1.chave, \ldots, c_x.chave$, seriam concatenados em um único grande vão entre t_1 e t_2. Dessa forma, os vários grupos de células não-originais consecutivas, que estavam inseridos nos vãos concatenados, iriam formar um grande grupo de células não-originais

consecutivas. Assim, precisamos estimar o tamanho de uma seqüência qualquer de vãos do instante t_1 consecutivos concatenados entre t_1 e t_2.

Lema 2. *O tamanho esperado de uma seqüência qualquer de células originais, consecutivas em uma lista ligada qualquer no instante t_1, que são removidas do* skip graph *entre t_1 e t_2 é $O(1)$.*

Demonstração. Seja c uma célula original de uma lista ligada qualquer do *skip graph*. Seja R o número de células originais que sucedem c, incluindo c, na lista ligada em questão no instante t_1, que foram removidas entre t_1 e t_2. Como a quantidade de remoções é no máximo a metade da quantidade de células originais, a probabilidade de que uma célula original qualquer seja removida é $\leq 1/2$ e $\Pr[R = x] \leq (1/2)^x$. Desta forma,

$$E[R] = \sum_{x=0}^{\infty} x \Pr[R = x] \leq \sum_{x=0}^{\infty} x(1/2)^x = O(1),$$

pelo mesmo cálculo feito anteriormente. □

Portanto, podemos calcular o tamanho esperado de uma seqüência qualquer de células não-originais consecutivas no *skip graph* entre t_1 e t_2:

Lema 3. *O tamanho esperado de uma seqüência qualquer de células não-originais consecutivas no* skip graph *entre t_1 e t_2 é $O(1)$.*

Demonstração. Entre t_1 e t_2, pelo lema 1, o número esperado de células não-originais consecutivas inseridas em um vão do instante t_1 de uma lista ligada qualquer é $O(1)$, e pelo lema 2, o tamanho esperado de uma seqüência qualquer de células originais consecutivas em uma lista ligada qualquer no instante t_1 que são removidas do *skip graph* é $O(1)$. Como essas variáveis aleatórias são independentes, o tamanho esperado de uma seqüência qualquer de células não-originais consecutivas no *skip graph* é $O(1) \cdot O(1) = O(1)$. □

4.4.1 Busca

O algoritmo 31 é como o algoritmo tradicional correspondente, salvo os comandos de obtenção e de liberação de *locks*. As inserções e remoções concorrentes são feitas sobre células com palavras aleatórias uniformemente distribuídas em Σ^*, já que supomos que eventuais adversários não possuem acesso a esses campos. Logo, podemos assumir que, excluindo o tempo de obtenção e liberação de *locks*, as buscas ainda gastam tempo esperado $O(\lg n)$, onde n é o maior número de células presentes no *skip graph* no decorrer da operação. Observe que tal conclusão independe da distribuição probabilística dos valores das chaves nas inserções e remoções de células na estrutura.

4.4.2 Inserção

Consideremos uma inserção qualquer, feita entre t_1 e t_2, nas condições definidas anteriormente. Além disso, consideremos n como o maior número de células presentes no *skip graph* no decorrer da operação. No algoritmo `inserir()` da página 49, temos apenas operações que gastam tempo constante até a linha 36, com exceção de uma busca na linha 11. Portanto, essas linhas gastam tempo esperado $O(\lg n)$.

No algoritmo `determinarVizinhosCompatAcima()`, temos apenas operações que gastam tempo constante, com exceção das duas chamadas a `encontrarVizinhoCompatAcima()`. Vamos então considerar que partimos de uma célula c_1 e procuramos, no nível j, por células compatíveis no nível $j + 1$. Como vimos no capítulo anterior, para encontrar uma célula compatível, em qualquer direção, é preciso visitar lateralmente um número esperado de $O(1)$ células. Desta forma, em `encontrarVizinhoCompatAcima()`, se não ignoramos células não inseridas no nível acima (com parâmetro $i =$ false), vamos gastar tempo esperado $O(1)$, e encontrar a primeira célula compatível c_2, esteja inserida no nível acima ou não.

Em contrapartida, se ignoramos células não inseridas no nível acima (com parâmetro $i =$ true), devemos ignorar células compatíveis $c_2, \ldots, c_{x-1}$ que não estejam inseridas no nível $j + 1$ quando fossem visitadas. Uma célula original com uma vizinha no nível j tem nível obrigatoriamente $\geq j+1$, e como essa célula já terminou seu processo de inserção no instante t_1, no instante em que ela é visitada ela já está certamente inserida no nível $j + 1$. Já uma célula não-original com uma vizinha de nível j já poderia estar inserida no nível $j + 1$ ou não. Desta forma: (1) as células ignoradas são necessariamente não-originais; (2) nunca seria ignorada uma célula original; e (3) o tamanho esperado do maior grupo de células não-originais consecutivas é $O(1)$ (pelo lema 3); podemos concluir que o número de células visitadas pelo algoritmo `encontrarVizinhoCompatAcima()`, até acharmos e aceitarmos uma célula c_x, é $O(1)$.

Assim, tanto com $i =$ false quanto com $i =$ true, o número esperado de células visitadas pelo algoritmo `encontrarVizinhoCompatAcima()` é de $O(1)$. Quando achamos c_x, ainda vamos chamar o algoritmo `encontrarMelhorVizinho()`, caminhando na direção oposta, no nível $j + 1$, em busca de células entre c_x e c_1 inseridas em $j + 1$ no decorrer da execução do primeiro algoritmo. O número esperado de células visitadas por `encontrarMelhorVizinho()` é o número esperado de células não-originais inseridas entre c_x e c_1, compatíveis com c_x e c_1, e é menor que o número esperado de células não-originais inseridas entre c_x e c_1, sem considerar compatibilidade. Esse número é $O(1)$, pois o tamanho esperado do maior grupo de células não-originais consecutivas é $O(1)$.

Desta forma, no algoritmo `encontrarVizinhoCompatAcima()`, o número esperado de células visitadas é $O(1)$, com $i =$ false ou $i =$ true, e então, no algoritmo `determinarVizinhosCompatAcima()`,

o número esperado de células visitadas é também $O(1)$. Com isso, concluímos que em cada nível vamos gastar tempo esperado $O(1)$.

Já que o tempo gasto esperado é $O(1)$ em cada nível, mesmo que `tentarPromoverLocks()` precise ser invocada $O(1)$ vezes para conseguir promover *locks* em cada nível, podemos observar que o algoritmo ainda gastaria tempo esperado $O(1)$ em cada nível. Já que a quantidade esperada de níveis nos quais uma célula é inserida é $O(\lg n)$, o algoritmo `inserir()` também gasta tempo esperado $O(\lg n)$.

Apesar das hipóteses fazerem com que as inserções gastem tempo esperado $O(\lg n)$, a inserção pode gastar tempo $\Omega(n)$ no pior caso. Por exemplo, vamos supor que todas as $s(t_1)$ inserções sejam feitas no mesmo vão de S_ϵ, e que antes da primeira célula se inserir no nível 1, todas as células são inseridas no nível 0. Nesse caso, o número de células não-originais consecutivas em um vão do instante t_1 é $\Theta(n)$.

4.4.3 Remoção

Consideremos uma remoção qualquer, feita entre t_1 e t_2, nas condições definidas anteriormente. Além disso, consideremos n como o maior número de células presentes no *skip graph* no decorrer da operação. No algoritmo `remover()` da página 58, temos apenas operações que gastam tempo constante até a linha 17, com exceção da busca feita na linha 4. Portanto, essas linhas gastam tempo esperado $O(\lg n)$. No laço da linha 17, temos apenas operações que gastam tempo constante. Na linha 33, o nível da célula em processo de remoção decresce, e não cresce novamente porque o último símbolo de sua palavra aleatória foi tornado inválido na linha 11 de `atribuirVizinho()`. Desta forma, em cada nível vamos gastar tempo esperado $O(1)$ (já considerando $O(1)$ invocações de `tentarPromoverLocks()` em cada nível, como na inserção), e no algoritmo completo, já que uma célula possui nível esperado $O(\lg n)$, vamos gastar tempo esperado $O(\lg n)$.

Capítulo 5

Desempenho e Aplicação dos *Skip Graphs*

Nesse capítulo, vamos fazer uma análise prática do desempenho dos *skip graphs* concorrentes, e em seguida mostramos uma possível aplicação de tais estruturas em sistemas distribuídos.

5.1 Desempenho dos *Skip Graphs* Concorrentes

No decorrer deste trabalho, foram feitas implementações dos algoritmos tradicionais e concorrentes apresentados, na linguagem Java. Nessa seção, vamos fazer experimentos práticos com esses algoritmos. Os diversos processos serão simulados através de múltiplas *threads* concorrentes, e os *locks* serão providos pela linguagem, na classe `ReadWriteLock` do pacote `java.util.concurrent`.

O objetivo dessa seção é verificar, em contextos experimentais específicos, as seguintes medidas:

1. O tempo médio de contenção dos diversos processos (o tempo médio em que os diversos processos consomem bloqueados – veja seção 4.4) nas invocações a `obterLock()`. No capítulo anterior, assumimos que o tempo gasto em `obterLock()` era independente do número de células e de processos envolvidos.

2. Quantas vezes, em média, é preciso invocar `tentarPromoverLocks()`, em cada nível da inserção ou da remoção, de modo a conseguir promover de fato os *locks*. No capítulo anterior, tal quantidade foi considerada independente do número de células e de processos envolvidos.

Para isso, foram feitos alguns testes, e 6 desses testes foram escolhidos para representar o que foi observado nos outros testes. Em cada teste, temos um processo especial que cria um *skip graph* com uma única célula e gera, em ciclos, vários outros processos que acessam concorrentemente a estrutura. Cada processo executa uma única operação sobre o *skip graph*: ou uma inserção ou uma remoção (adiante veremos quantos executam inserção e quantos executam remoção). Cada ciclo consiste na geração de um número de processos por parte do processo especial e na execução de operações concorrentes por parte dos processos lançados.

O processo especial mencionado anteriormente define certos parâmetros que caracterizam cada teste. Os parâmetros que consideramos são: (1) o tamanho do conjunto das possíveis chaves, $|U|$ (o maior tamanho possível do *skip graph*); (2) a razão entre o número de processos gerados a cada ciclo e o tamanho corrente do *skip graph*, γ; (3) o número de inserções concorrentes bem-sucedidas a serem realizadas, I; e (4) o número de remoções concorrentes bem-sucedidas a serem realizadas, R. O segundo parâmetro é doravante denominado *grau de concorrência*. Acreditamos que os parâmetros anteriores sejam os principais parâmetros que poderiam influir nos valores que estamos medindo.

Fixados esses parâmetros, o processo especial realiza vários ciclos de criação de novos processos, até que I inserções bem-sucedidas e R remoções bem-sucedidas tenham sido feitas. No início do x-ésimo ciclo, digamos iniciado no instante t_1, lançamos I_x inserções e R_x remoções, onde $I_x = \gamma \cdot n(t_1)$ e $R_x = \gamma \cdot n(t_1)/2$, a não ser que hajam menos que $\gamma \cdot n(t_1)$ inserções e $\gamma \cdot n(t_1)/2$ remoções que devam ser feitas com sucesso para completarmos as I e R inserções e remoções almejadas. Se isso ocorrer, lançamos somente o número de inserções e de remoções que devem ser feitas para isso. Desta forma, é possível perceber que nos ciclos finais temos uma quantidade reduzida de operações concorrentes. Isso está relacionado com o fato de que várias inserções e remoções são inócuas em cada ciclo, já que, quando essas operações forem realizadas, as chaves a serem inseridas podem estar presentes no *skip graph* ou as chaves a serem removidas podem estar ausentes do *skip graph*.

Nos nossos testes, vamos sempre tomar $0 < \gamma \leq 1$. Desta forma, $I_x \leq n(t_1)$ e $R_x \leq n(t_1)/2$, o que permite contemplar a hipótese de que o tamanho da estrutura é no máximo duplicado ou reduzido à metade em cada ciclo de inserções e de remoções. As comparações serão feitas em um sistema monoprocessado, onde não existe nenhum ganho com paralelismo. Esse ambiente experimental já é suficiente para observarmos o tempo de contenção de processos e o comportamento das diversas promoções de *locks*.

Ao final de todos os ciclos, nossa estrutura contém exatamente $I - R + 1$ células. As chaves usadas na inserção e remoção são uniformemente distribuídas em $\{1, \ldots, |U|\}$, onde $|U|$ é o primeiro parâmetro do teste em questão, como mencionamos anteriormente. Ademais, para cada inserção ou remoção, a célula tomada como apresentadora é escolhida aleatoriamente dentre as células presentes no instante do lançamento das operações.

Os testes foram feitos todos em um computador pessoal com processador AMD Athlon 64$^{\text{TM}}$ 3000+ (2 GHz), 512 MB de memória principal e sistema operacional Microsoft Windows$^{\text{TM}}$. A versão do compilador e do interpretador Java é 1.5, e o *profiler* empregado a fim de obter os dados de execução é o JProfiler$^{\text{TM}}$. Esse *profiler* contabiliza o tempo gasto em cada função e o número de invocações realizadas a cada função, além de monitorar os estados dos processos envolvidos e de gerar gráficos.

Em três dos testes considerados, $I = 250$, $R = 200$ e $|U| = 350$. Cada um deles foi feito com um certo grau de concorrência, tomado dentre 10%, 50% e 100%. Em outros três dos testes considerados, $I = 65$, $R = 50$ e $|U| = 75$, cada um deles feito também com um certo grau de concorrência, tomado dentre 10%, 50% e 100%. O alfabeto utilizado nas palavras aleatórias do *skip graph* foi sempre binário. Todos esses números foram fixados de acordo com a capacidade computacional da máquina utilizada para fazer as diversas simulações[1]. Visando aumentar a acurácia dos resultados, foram feitas 10 simulações para cada um dos 6 testes, e o resultado de cada teste foi obtido da média de dados gerados pelas 10 simulações correspondentes.

Inicialmente, vamos fixar $I = 250$, $R = 200$ e $|U| = 350$, e fazer γ assumir os valores 10%, 50% e 100%. Os resultados são vistos nas tabelas 5.1, 5.2 e 5.3, e nas figuras 5.1, 5.2 e 5.3.

As tabelas seguintes possuem duas partes. Na primeira parte da tabela, na coluna 1, destacamos algoritmos importantes que são chamados por `inserir()` e `remover()`. Na coluna 2, destacamos a porcentagem do tempo gasto nas execuções das chamadas `inserir()` do teste em questão, relativas a cada um dos algoritmos importantes mencionados anteriormente. Na coluna 3, o mesmo é feito com relação ao tempo gasto nas execuções das chamadas `remover()`. Ou seja, na simulação descrita na tabela 5.1, em média quase 47% do tempo gasto em operações de remoção é gasto em execuções do algoritmo `atribuirVizinho()`.

Na segunda parte da tabela, temos a média do tempo total gasto nas 10 simulações do teste em questão, e a média da quantidade de operações de inserção e remoção lançadas para completar as I e R operações estipuladas, respectivamente. Essas quantidades podem ultrapassar I e R porque podemos iniciar inserções para elementos que já estão presentes no *skip graph* ou iniciar remoções para elementos que já estão em processo de remoção do *skip graph*. Em seguida, temos a média e o desvio padrão do número de invocações necessárias a `tentarPromoverLocks()` (linha 6 em `inserirNiveisSuperiores()` e linha 25 em `remover()`) em cada grupo de iterações iniciadas com um mesmo valor de j nesses laços (ou seja, iterações iniciadas em um mesmo nível da inserção ou da remoção no *skip graph*).

Já nas figuras, a abcissa representa tempo, e a ordenada representa quantidade de processos em execução concorrente. As *threads* são divididas em: (1) *threads* em execução; (2) *threads* bloqueadas; e (3) *threads* em espera. As últimas se referem às *threads* já terminadas mas ainda não limpas da memória do programa, e são mais visíveis somente nas três últimas figuras. De qualquer maneira, não são alvo de nosso interesse.

Em seguida, vamos fixar $I = 65$, $R = 50$ e $|U| = 75$, e fazer γ assumir os valores 10%, 50% e 100%, como feito antes. Os resultados são vistos nas tabelas 5.4, 5.5 e 5.6, e nas figuras 5.4, 5.5 e

[1] De forma que o sistema operacional não fizesse *swapping* de memória em disco, dentre outros fatores.

Operação	% do tempo `inserir()`	% do tempo `remover()`
obterLocks()	5.68	7.18
buscar()	8.61	18.96
atribuirVizinho()	36.18	46.78
determinarVizinhosCompatAcima()	24.18	-

Tempo total da simulação (em s)	12.6
Número de inserções lançadas	351.4
Número de remoções lançadas	721.4
Número médio e desvio padrão de promoções por nível (inserção)	1.159 / 0.013
Número médio e desvio padrão de promoções por nível (remoção)	1.267 / 0.004

Tabela 5.1: *Dados obtidos nas operações, com* $\gamma = 10\%$, $I = 250$, $R = 200$ *e* $|U| = 350$.

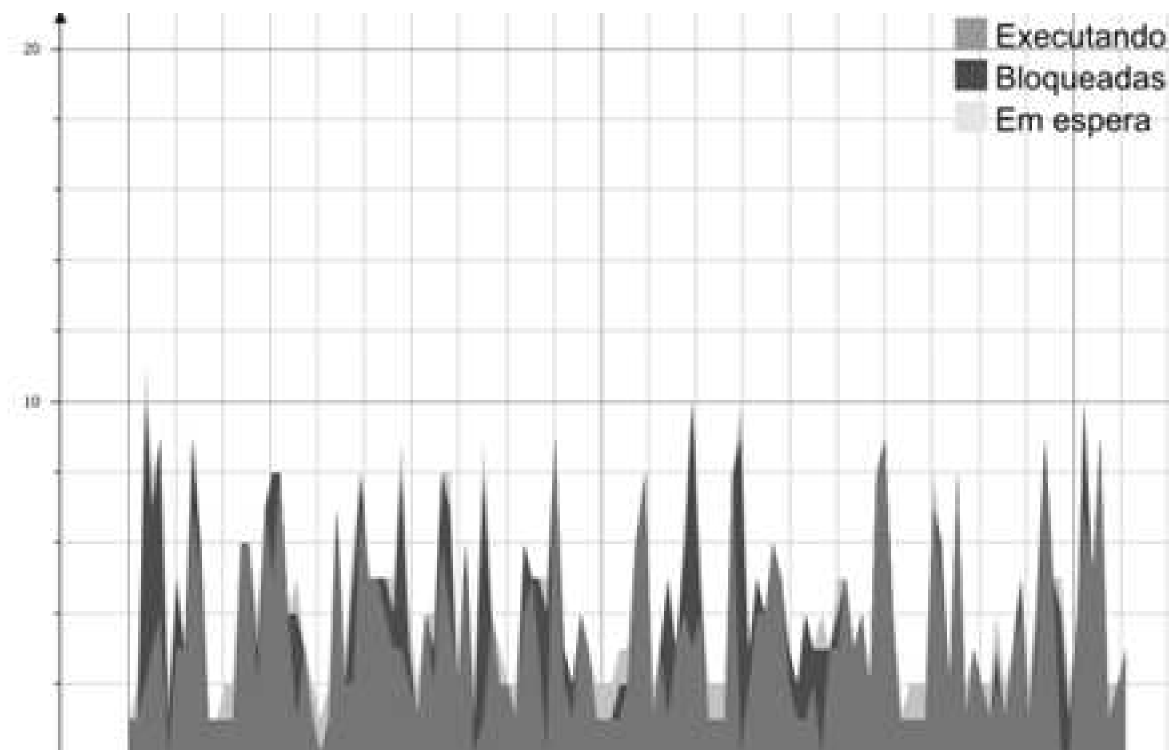

Figura 5.1: *Estado dos processos no decorrer da execução com* $\gamma = 10\%$, $I = 250$, $R = 200$ *e* $|U| = 350$.

5.6.

Com relação aos resultados apresentados, é possível perceber que o número médio de invocações a `tentarPromoverLocks()`, em cada nível da inserção e da remoção, não variou muito à medida que o grau de concorrência aumentou. O número médio de invocações a `tentarPromoverLocks()` não aumentou à medida que o tamanho do *skip graph* aumentou: na verdade, o número médio de tais

Operação	% de `inserir()`	% de `remover()`
obterLocks()	3.80	6.71
buscar()	8.87	21.21
atribuirVizinho()	25.20	39.94
determinarVizinhosCompatAcima()	18.98	-

Tempo total da simulação (em s)	14.1
Número de inserções lançadas	353.9
Número de remoções lançadas	776.7
Número médio e desvio padrão de promoções por nível (inserção)	1.172 / 0.015
Número médio e desvio padrão de promoções por nível (remoção)	1.272 / 0.006

Tabela 5.2: *Dados obtidos nas operações, com* $\gamma = 50\%$*,* $I = 250$*,* $R = 200$ *e* $|U| = 350$*.*

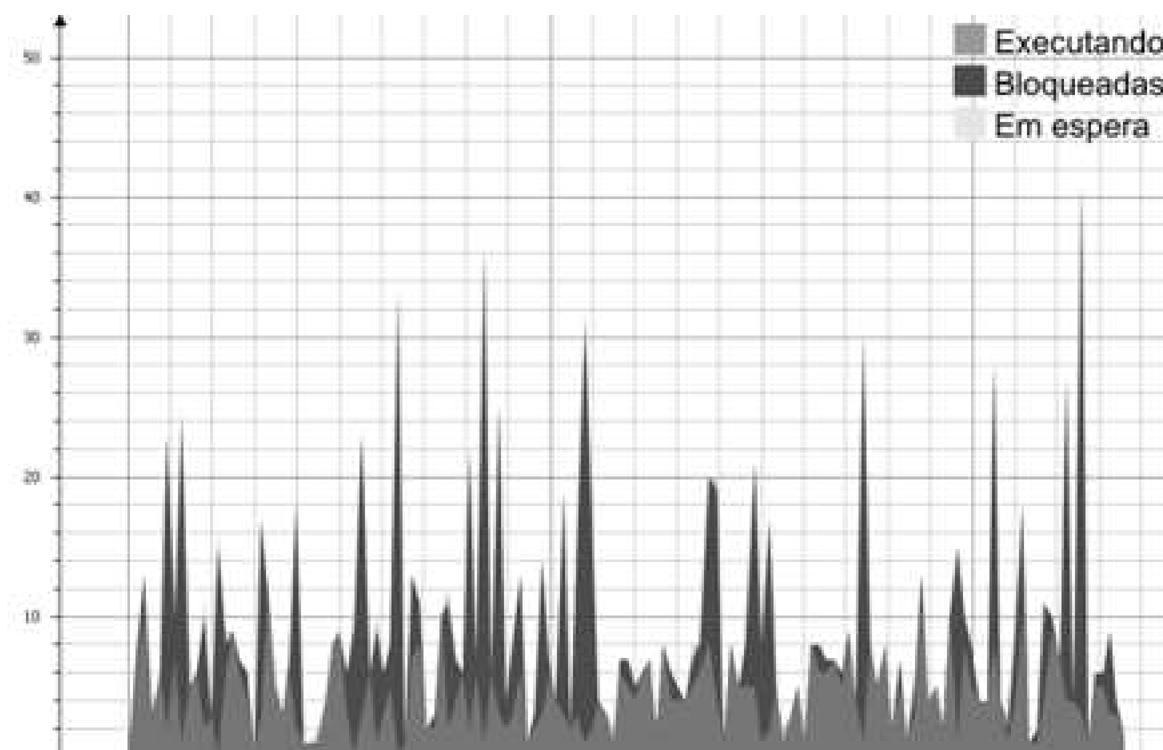

Figura 5.2: *Estado dos processos no decorrer da execução com* $\gamma = 50\%$*,* $I = 250$*,* $R = 200$ *e* $|U| = 350$*.*

invocações é menor nos três primeiros resultados (com estruturas maiores) e maior nos três últimos resultados (com estruturas menores), tanto para a inserção quanto para a remoção. Dessa forma, nossos testes básicos não parecem refutar a hipótese anterior, de que o número médio de invocações a `tentarPromoverLocks()`, em cada nível da inserção e da remoção, é $O(1)$.

Da mesma forma, o tempo gasto em `obterLocks()` não parece estar relacionado com quaisquer

Operação	% de `inserir()`	% de `remover()`
obterLocks()	2.61	6.08
buscar()	10.47	24.95
atribuirVizinho()	18.60	36.93
determinarVizinhosCompatAcima()	18.76	-

Tempo total da simulação (em s)	14.7
Número de inserções lançadas	357.9
Número de remoções lançadas	758.0
Número médio e desvio padrão de promoções por nível (inserção)	1.170 / 0.006
Número médio e desvio padrão de promoções por nível (remoção)	1.271 / 0.004

Tabela 5.3: *Dados obtidos nas operações, com* $\gamma = 100\%$*,* $I = 250$*,* $R = 200$ *e* $|U| = 350$*.*

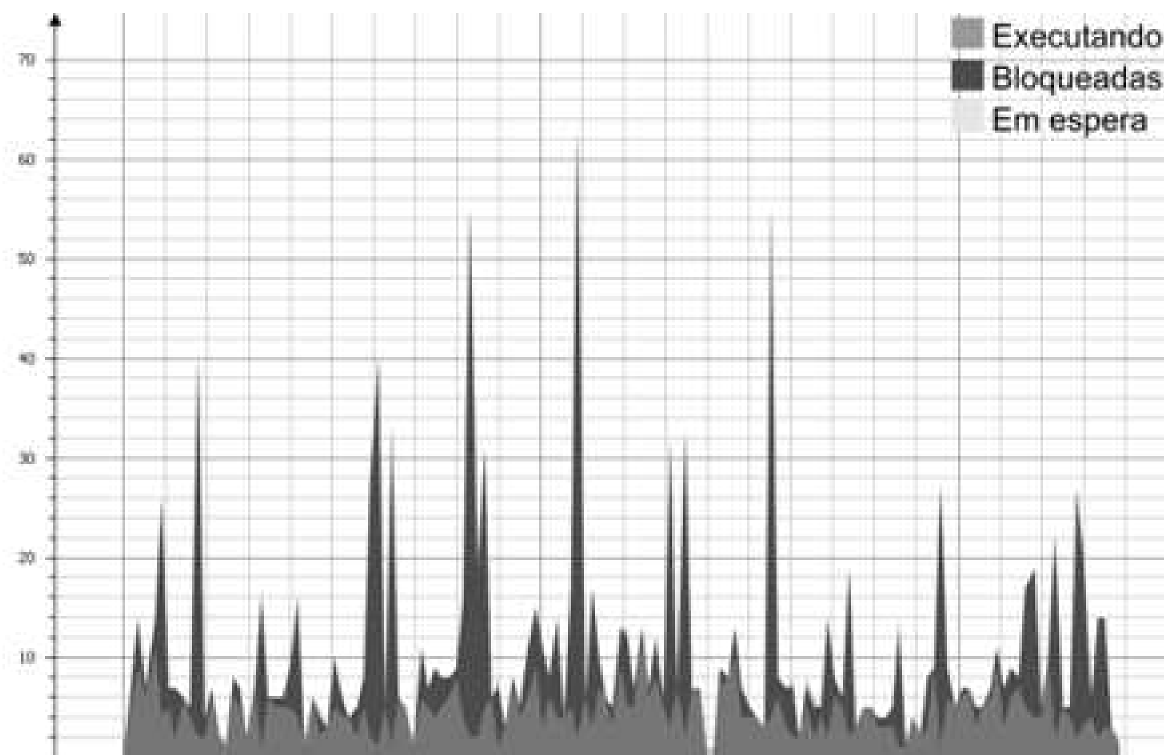

Figura 5.3: *Estado dos processos no decorrer da execução com* $\gamma = 100\%$*,* $I = 250$*,* $R = 200$ *e* $|U| = 350$*.*

parâmetros variados. O tempo gasto é maior quando o *skip graph* é menor e quando o grau de contenção é maior. O motivo pelo qual isso ocorre não está muito claro a princípio, e análises práticas mais detalhadas poderiam elucidá-lo. Todavia, estes dados não se direcionam no sentido de comprometer nossa hipótese de que as invocações a `obterLocks()` gastam tempo $O(1)$.

Operação	% do tempo `inserir()`	% do tempo `remover()`
obterLocks()	7.05	10.45
buscar()	5.69	15.28
atribuirVizinho()	54.01	63.42
determinarVizinhosCompatAcima()	20.20	-

Tempo total da simulação (em s)	0.62
Número de inserções lançadas	320
Número de remoções lançadas	706
Número médio e desvio padrão de promoções por nível (inserção)	1.237 / 0.019
Número médio e desvio padrão de promoções por nível (remoção)	1.463 / 0.034

Tabela 5.4: *Dados obtidos nas operações, com* $\gamma = 10\%$, $I = 65$, $R = 50$ *e* $|U| = 75$.

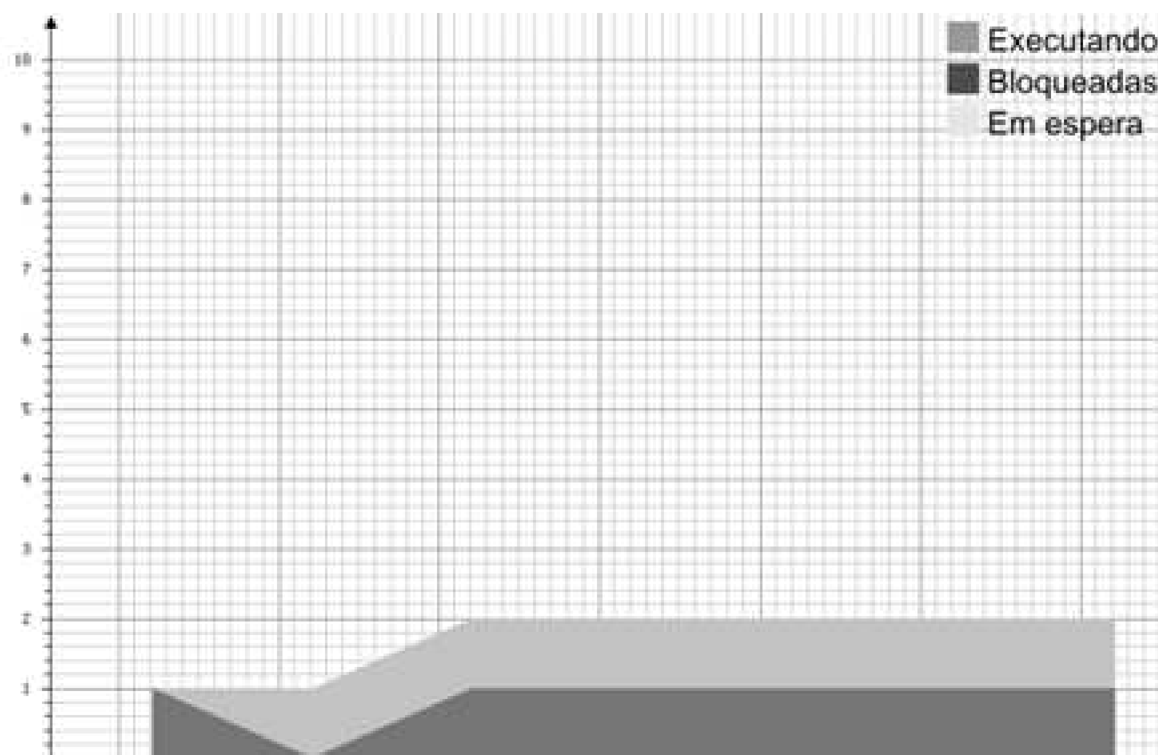

Figura 5.4: *Estado dos processos no decorrer da execução com* $\gamma = 10\%$, $I = 65$, $R = 50$ *e* $|U| = 75$.

5.2 Aplicação em Sistemas Distribuídos

Nessa seção, vamos falar a respeito da aplicação de estruturas como árvores balanceadas de busca, *skip lists* e *skip graphs* em ambientes distribuídos. Começaremos primeiramente com uma discussão genérica a respeito de sistemas distribuídos.

Muitos dos sistemas computacionais existentes atualmente são *sistemas distribuídos*, constituídos

Operação	% de `inserir()`	% de `remover()`
obterLocks()	5.01	5.47
buscar()	10.77	12.74
atribuirVizinho()	25.09	29.53
determinarVizinhosCompatAcima()	13.98	-

Tempo total da simulação (em s)	0.83
Número de inserções lançadas	378
Número de remoções lançadas	751
Número médio e desvio padrão de promoções por nível (inserção)	1.253 / 0.032
Número médio e desvio padrão de promoções por nível (remoção)	1.519 / 0.083

Tabela 5.5: *Dados obtidos nas operações, com* $\gamma = 50\%$, $I = 65$, $R = 50$ *e* $|U| = 75$.

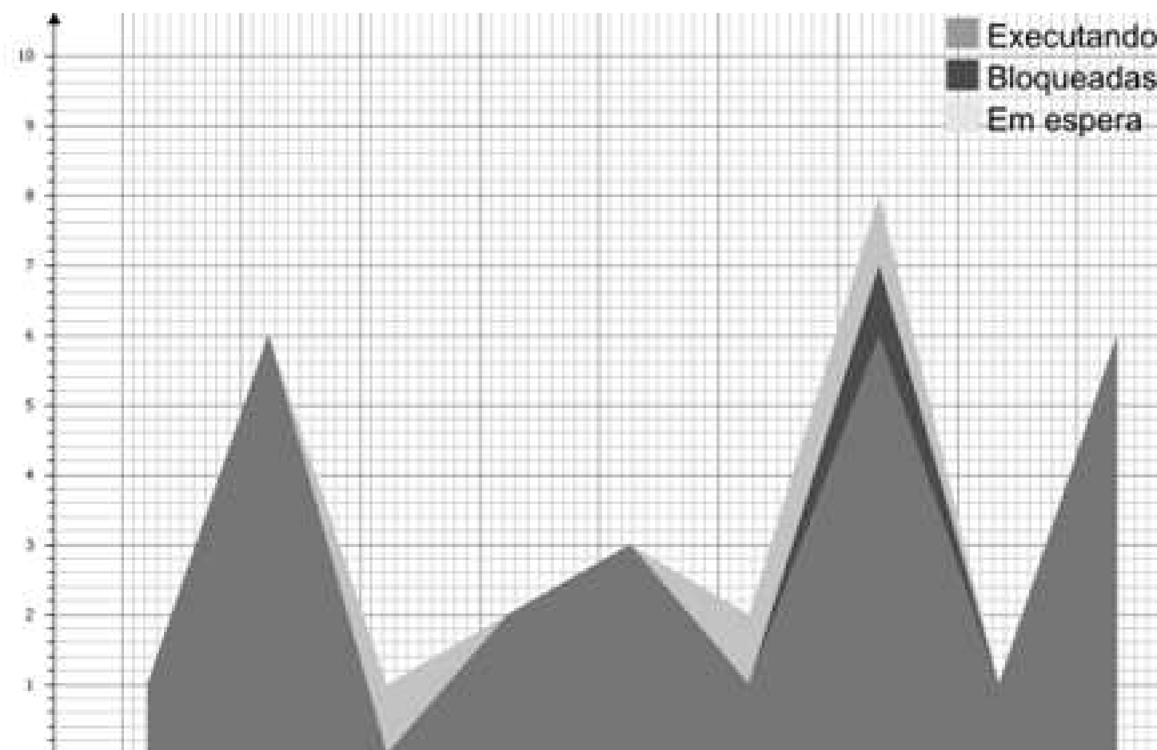

Figura 5.5: *Estado dos processos no decorrer da execução com* $\gamma = 50\%$, $I = 65$, $R = 50$ *e* $|U| = 75$.

por diversos *processos* em execução em diversas máquinas. Um processo é identificado inambiguamente por um "endereço completo", que consiste tipicamente do endereço IP da máquina associada e do número da porta TCP/UDP no sistema operacional dessa máquina. As conexões físicas entre as máquinas permitem conexões lógicas entre os processos. Para haver uma conexão entre processos, é necessário e suficiente que um deles possua uma *referência* para o outro processo. Uma referência é simplesmente o endereço completo de um processo.

Operação	% de `inserir()`	% de `remover()`
obterLocks()	3.18	4.49
buscar()	8.45	15.82
atribuirVizinho()	16.73	21.84
determinarVizinhosCompatAcima()	11.12	-

Tempo total da simulação (em s)	0.91
Número de inserções lançadas	369
Número de remoções lançadas	671
Número médio e desvio padrão de promoções por nível (inserção)	1.278 / 0.032
Número médio e desvio padrão de promoções por nível (remoção)	1.545 / 0.061

Tabela 5.6: *Dados obtidos nas operações, com $\gamma = 100\%$, $I = 65$, $R = 50$ e $|U| = 75$.*

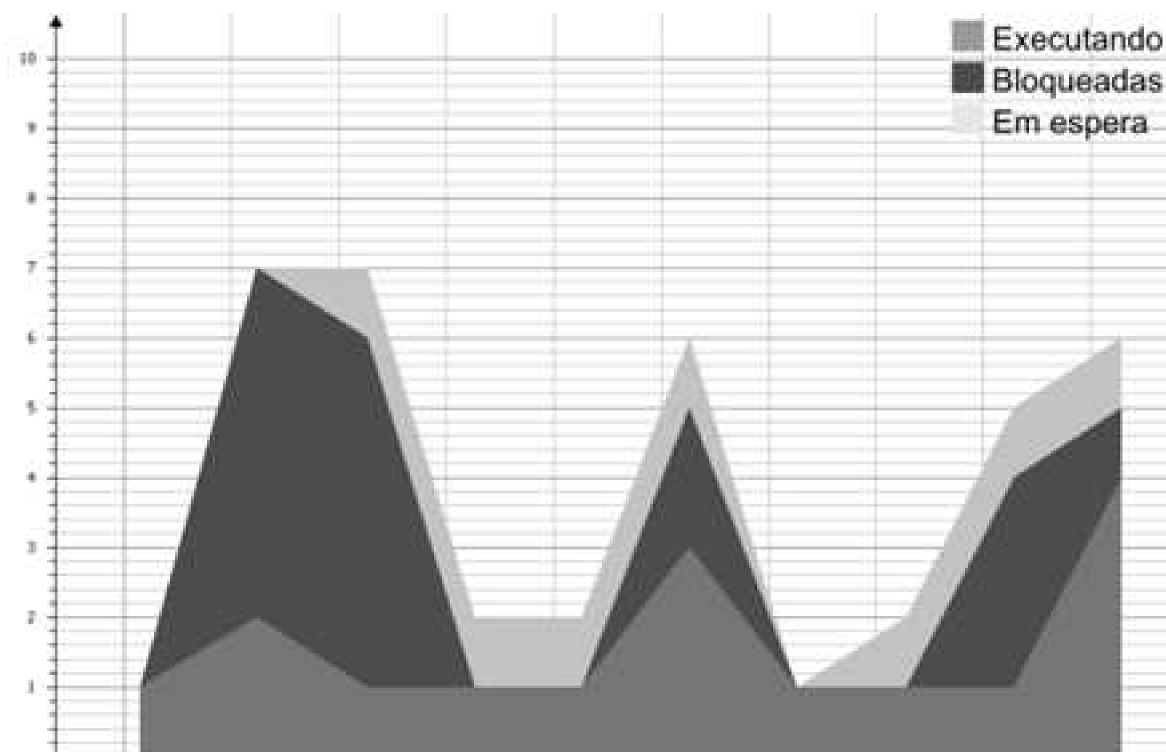

Figura 5.6: *Estado dos processos no decorrer da execução com $\gamma = 100\%$, $I = 65$, $R = 50$ e $|U| = 75$.*

Para simplificar, os processos serão denotados por nomes, que identificam inambiguamente os processos correspondentes, e as referências serão apenas nomes de processos.

Dentre os sistemas distribuídos, podemos destacar os sistemas *peer-to-peer*. Nestes sistemas, os processos se auto-organizam no sentido de estabelecer as referências entre eles. Mais especificamente, as conexões entre os processos são estabelecidas sem qualquer autoridade externa, pois cada processo

é responsável por procurar outros processos do sistema distribuído e se conectar a eles. Além disso, os processos devem reagir às variáveis e voláteis condições do sistema distribuído, sempre interagindo, mais uma vez, sem qualquer autoridade externa [3].

Dentre as operações fundamentais de muitos sistemas distribuídos, e em particular de sistemas *peer-to-peer*, estão a inserção, remoção e localização de processos. Já que o tamanho desses sistemas, medido no número de processos, é normalmente muito grande, uma abordagem ingênua para implementar essas operações não seria tão eficiente, principalmente no que se refere à localização. Portanto, poderíamos utilizar estruturas de dados como árvores balanceadas de busca, *skip lists* ou *skip graphs* para tornar essas operações bem mais eficientes. Nesse caso, as células da estrutura seriam agora os processos e os apontadores da estrutura. Quando uma estrutura de dados é empregada desta forma, vamos dizer que temos uma *estrutura de dados distribuída*.

Todavia, as premissas e os requisitos de um ambiente distribuído são diferentes dos correspondentes em um ambiente multiprocessado. Além da concorrência, podemos destacar:

Distribuição de Carga: Em uma estrutura de dados distribuída, um dado acessado em todas as operações, como seria a raiz de uma árvore balanceada de busca ou a cabeça de uma *skip list*, é um fator bastante indesejável, pois a máquina associada a este dado ficaria facilmente sobrecarregada;

Tolerância a Falhas: Em um sistema distribuído, os processos podem falhar, em conseqüência de falhas físicas nas máquinas associadas ou na rede de comunicação entre essas máquinas. Isso pode fazer com que algumas referências de uma estrutura de dados distribuída se tornem erradas. Além disso, são necessários tratamentos adicionais nos mecanismos de obtenção e liberação de *locks*, já que as falhas de processos poderiam implicar em *deadlocks* (se um recurso obtido sumisse do sistema porque falhou o processo associado, por exemplo).

A primeira diferença praticamente inviabiliza a aplicação de árvores balanceadas de busca e de *skip lists* em um ambiente distribuído. Os *skip graphs* não possuem cabeça, e não estão sujeitos a esse problema. A segunda diferença já nos traz diversas complicações. Se quisermos empregar os *skip graphs* em um ambiente distribuído, precisamos empregar algoritmos que "consertem" inconsistências que possam surgir na estrutura. Aspnes e Shah [6] propuseram alguns algoritmos que visam essa tarefa. Além disso, são necessários mecanismos diferentes para controlar a concorrência entre os diversos processos, já que o uso de *locks* sem medidas adicionais poderia implicar na ocorrência de *deadlocks* em tais ambientes. Essas preocupações estão fora do escopo desse texto, sendo trabalhos futuros apropriados e interessantes.

5.2.1 Compartilhamento Distribuído de Arquivos

Nessa seção, vamos mostrar como *skip graphs* concorrentes, se adaptados para tolerar falhas, poderiam embasar uma aplicação muito comum de sistemas *peer-to-peer*: o compartilhamento distribuído de arquivos. Vamos assumir que cada máquina estaria associada a uma célula do *skip graph* e vice-versa. Desta forma, o *skip graph* estaria distribuído entre diversas máquinas, e cada uma dessas máquinas atuaria como um processo no sistema distribuído.

Definimos inicialmente os seguintes conjuntos:

- P: todos os possíveis nomes de processos (seus endereços completos);
- A: todos os possíveis nomes de arquivos;
- I_m: $\{0, \ldots, 2^m - 1\}$, com $m \geq 0$ fixado previamente.

Dizemos que P_t e A_t são os processos e arquivos, respectivamente, que estão presentes no sistema no instante t. Vamos definir que uma função de identificação é uma função de espalhamento f tal que $f : P \to I_m$ ou $f : A \to I_m$.

Vamos então mostrar como as máquinas e os arquivos são dispostos no sistema distribuído. Cada célula do *skip graph* teria chave $= id_p(p)$, onde $id_p : P \to I_m$ é uma função de identificação fixada previamente, e p é o nome (endereço completo) do processo associado à célula. Chamamos $id_p(p)$ de identificador do processo p. Além disso, cada arquivo seria associado a um número $x \in \{1, \ldots, |U|\}$, por meio de uma outra função de identificação fixada previamente, $id_a(a) : A \to I$, onde a é o nome do arquivo mapeado. Chamamos $id_a(a)$ de identificador do arquivo a. Um arquivo a seria disposto no sistema distribuído da seguinte maneira: $i = id_a(a)$ seria calculado, e o arquivo seria guardado na máquina cuja célula associada primeiro seguisse i em S_ϵ. Assim, quando houvesse inserções, alguns arquivos deveriam ser remanejados de processos. Desta forma, se quiséssemos localizar um arquivo, bastaria procurar pelo identificador do arquivo no *skip graph*, e, conforme sabemos, ou encontraríamos a célula associada ao arquivo procurado, ou uma célula precedente à célula associada ao arquivo procurado.

Na abordagem apresentada, podemos perceber que as chaves das diversas células do *skip graph* possuem valores uniformemente distribuídos em $\{1, \ldots, |U|\}$. Na verdade, abordagens parecidas, que têm essa propriedade, são muito comuns. Por esse motivo, no capítulo anterior, foi dito que a hipótese de que chaves sejam uniformemente distribuídas em um espaço numérico qualquer não era tão irreal quanto parecia.

Capítulo 6

Considerações Finais

Os sistemas multiprocessados e distribuídos trazem muitos desafios à concepção de algoritmos e estruturas de dados apropriados para esses ambientes. O primeiro grande desafio, tópico discutido nesse texto, é o controle da concorrência entre processos. Nos nossos algoritmos, empregamos *locks* para tratar desse problema, mas existem outras abordagens possíveis. Na literatura, há algumas que utilizam diretamente primitivas como `CAS` para controlar a concorrência dos algoritmos, de forma que tais primitivas sempre causem "progresso" ao sistema, ao contrário dos *locks*. Essas abordagens são chamadas de *lock-free* [39,42], e em geral são bem mais complexas e mais eficientes do que abordagens baseadas em *locks*. Assim, se procura implementar versões *lock-free* para estruturas de dados simples e básicas, como listas e pilhas, entre outras [15, 28, 40, 41]. Essas estruturas seriam então utilizadas como base para implementar estruturas de dados mais complexas.

Existem ainda abordagens que possuem uma propriedade denominada *wait-freedom* [18], que garante também que qualquer processo termina sua operação em um número finito de passos, independentemente do comportamento dos outros processos envolvidos. Obviamente, isso implica em uma complexidade ainda maior, o que também torna essa propriedade mais restrita a estruturas de dados simples.

No nosso texto, foi mostrada uma estrutura de dados mais complexa que emprega uma abordagem de sincronização mais simples. Em sua implementação, percebemos a dificuldade para lidar com a concorrência mesmo usando *locks*. Além disso, pudemos perceber como a contenção de processos tem efeitos negativos visíveis em aplicações concorrentes. Um trabalho futuro interessante talvez fosse averiguar a possibilidade de aplicar versões *lock-free* de listas ligadas concorrentes na implementação de *skip graphs* ou mesmo de outras estruturas concorrentes. Outra possibilidade interessante seria avaliar abordagens diversas no tratamento de falhas na obtenção de *locks*, procurando identificar as melhores e piores abordagens.

Bibliografia

[1] I. Abraham, J. Aspnes, and J. Yuan, *Skip B-trees*, Ninth International Conference on Principles of Distributed Systems (pre-proceedings), December 2005, pp. 284–295.

[2] G. M. Adelson-Velskii and E. M. Landis, *An algorithm for the organization of information.*, Proceedings of the USSR Academy of Sciences, vol. 146, 1962, pp. 263–266.

[3] S. Androutsellis-Theotokis and D. Spinellis, *A survey of peer-to-peer content distribution technologies*, ACM Computing Surveys **36** (2004), no. 4, 335–371.

[4] L Arge, D Eppstein, and M. T. Goodrich, *Skip-webs: efficient distributed data structures for multi-dimensional data sets*, PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of Distributed Computing (New York, NY, USA), ACM Press, 2005, pp. 69–76.

[5] J. Aspnes, J. Kirsch, and A. Krishnamurthy, *Load balancing and locality in range-queriable data structures*, Twenty-Third ACM Symposium on Principles of Distributed Computing, July 2004, pp. 115–124.

[6] J. Aspnes and G. Shah, *Skip graphs*, SODA '03: Proceedings of the fourteenth annual ACM-SIAM Symposium on Discrete Algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 2003, pp. 384–393.

[7] B. Awerbuch and C. Scheideler, *Peer-to-peer systems for prefix search*, PODC '03: Proceedings of the twenty-second annual ACM symposium on Principles of Distributed Computing (New York, NY, USA), ACM Press, 2003, pp. 123–132.

[8] E. G. Coffman, M. Elphick, and A. Shoshani, *System deadlocks*, ACM Comput. Surv. **3** (1971), no. 2, 67–78.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 2 ed., The MIT Press, 2001.

[10] Intel Corporation, *Intel 64 and IA-32 architectures software developer's manual.*

[11] I. Foster and A. Iamnitchi, *On death, taxes, and the convergence of peer-to-peer and grid computing*, 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03) (Berkeley, CA, USA), February 2003.

[12] I. Foster, C. Kesselman, and S. Tuecke, *The anatomy of the grid: Enabling scalable virtual organizations*, International Journal of High Performance. Computing and Applications **15** (2001), no. 3, 200–222.

[13] P. B. Gibbons, *The asynchronous PRAM: a semisynchronous model for shared-memory MIMD machines*, Ph.D. thesis, University of California, Berkeley, Berkeley, CA, USA, 1989.

[14] M. Goodrich, M. Nelson, and J. Sun, *The rainbow skip graph: a fault-tolerant constant-degree distributed data structure*, SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete Algorithms (New York, NY, USA), ACM Press, 2006, pp. 384–393.

[15] T. L. Harris, *A pragmatic implementation of non-blocking linked-lists*, DISC '01: Proceedings of the 15th International Conference on Distributed Computing (London, UK), Springer-Verlag, 2001, pp. 300–314.

[16] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, *Skipnet: A scalable overlay network with practical locality properties*, In proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03) (Seattle, WA, USA), March 2003.

[17] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit, *A lazy concurrent list-based set algorithm*, Proc. of the 9th International Conference On Principles Of Distributed Systems (OPODIS 2005), 2005, pp. 3–16.

[18] M. Herlihy, *Wait-free synchronization*, ACM Transactions on Programming Languages and Systems (TOPLAS) **13** (1991), no. 1, 124–149.

[19] A. Iamnitchi and I. Foster, *A peer-to-peer approach to resource location in grid environments*, Grid Resource Management: State of the Art and Future Trends (2004), 413–429.

[20] ______, *A peer-to-peer approach to resource location in grid environments*, Grid Resource Management: State of the Art and Future Trends (2004), 413–429.

[21] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web*, ACM Symposium on Theory of Computing, May 1997, pp. 654–663.

[22] D. Karger and M. Ruhl, *Simple efficient load balancing algorithms for peer-to-peer systems*, SPAA '04: Proceedings of the sixteenth annual ACM Symposium on Parallelism in Algorithms and Architectures (New York, NY, USA), ACM Press, 2004, pp. 36–43.

[23] R. M. Karp, *A survey of parallel algorithms for shared-memory machines*, Tech. Report CSD-88-408, University of California at Berkeley, Berkeley, CA, USA, 1988.

[24] L. Lamport, *The mutual exclusion problem: Part I – the theory of interprocess communication*, Journal of the ACM **33** (1986), no. 2, 313–326.

[25] ______, *The mutual exclusion problem: Part II – statement and solutions*, Journal of the ACM **33** (1986), no. 2, 327–348.

[26] Y. Lev, M. Herlihy, V. Luchangco, and N. Shavit, *A provably correct scalable skiplist (brief announcement)*, Proc. of the 10th International Conference On Principles Of Distributed Systems (OPODIS 2006), 2006.

[27] G. S. Manku, M. Naor, and U. Wieder, *Know thy neighbor's neighbor: the power of lookahead in randomized p2p networks*, STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of Computing (New York, NY, USA), ACM Press, 2004, pp. 54–63.

[28] M. M. Michael and M. L. Scott, *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*, PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of Distributed Computing (New York, NY, USA), ACM, 1996, pp. 267–275.

[29] W. Pugh, *Skip lists: A probabilistic alternative to balanced trees*, Workshop on Algorithms and Data Structures, 1989, pp. 437–449.

[30] W Pugh, *Concurrent maintenance of skip lists*, Tech. Report CS-TR-2222, University of Maryland at College Park, 1990.

[31] W. Pugh, *A skip list cookbook*, Tech. Report CS-TR-2286.1, University of Maryland, College Park, 1990.

[32] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, *A scalable content-addressable network*, SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (New York, NY, USA), ACM Press, 2001, pp. 161–172.

[33] J. H. Reif, *Synthesis of parallel algorithms*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[34] M. Ripeanu, *Peer-to-peer architecture case study: Gnutella network*, P2P '01: Proceedings of the 1st International Conference on Peer-to-Peer Computing (P2P'01) (Washington, DC, USA), IEEE Computer Society, 2001, pp. 99–100.

[35] A. Rowstron and P. Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), November 2001, pp. 329–350.

[36] R. Sedgewick, *Algorithms in C – parts 1-4: Fundamentals, data structures, sorting, searching*, 3 ed., Addison-Wesley Professional, 1997.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (New York, NY, USA), ACM Press, 2001, pp. 149–160.

[38] ———, *Chord: A scalable peer-to-peer lookup service for internet applications*, Tech. Report TR-819, MIT, March 2001.

[39] R. K. Treiber, *Systems programming: Coping with parallelism*, Tech. Report RJ 5118, IBM Almaden Research Center, April 1986.

[40] J. D. Valois, *Implementing lock-free queues*, Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems (Las Vegas, NV), 1994, pp. 64–69.

[41] ———, *Lock-free linked lists using compare-and-swap*, PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of Distributed Computing (New York, NY, USA), ACM, 1995, pp. 214–222.

[42] ———, *Lock-free data structures*, Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1996.

[43] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*, Tech. Report UCB/CSD-01-1141, UC Berkeley, April 2001.