Estruturas de dados persistentes

Yan Soares Couto

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
Orientadora: Profa. Dra. Cristina Gomes Fernandes

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da FAPESP e CNPq.

São Paulo, novembro de 2018

Estruturas de dados persistentes

Esta é a versão original da dissertação elaborada pelo
candidato Yan Soares Couto, tal como
submetida à Comissão Julgadora.

Resumo

COUTO, Y. S. **Estruturas de dados persistentes**. 2018. 80 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Os problemas de ancestral comum mais profundo e ancestral de nível em árvores podem ser resolvidos com complexidade sub-quadrática. Estruturas de dados persistentes permitem acesso ou modificação a suas versões anteriores. Pilhas, filas e deques podem ser implementadas de forma persistente. Existem técnicas gerais para tornar alguns tipos de estruturas de dados persistentes. Árvores rubro-negras podem ser implementadas de forma persistente com espaço adicional constante por operação. O problema da localização de ponto pode ser resolvido utilizando uma árvore de busca binária balanceada persistente.

Palavras-chave: estruturas de dados, persistência, árvores rubro-negra, localização de ponto

Abstract

Couto, Y. S. **Persistent data structures**. 2018. 80 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

The lowest common ancestor and level ancestor problems can be solved in sub-quadratic complexity. Persistent data structures allow the access or modification of their previous versions. Stacks, queues and deques can be made persistent. There are general techniques to make certain types of data structures persistent. Red-black trees can be made persistent with constant additional space per operation. The point location problem can be solved using a persistent balanced binary search tree.

Keywords: data structures, persistence, red-black trees, point location

Sumário

Introdução

Uma estrutura de dados (ED) é uma forma de organizar dados em programas de computador. Estruturas de dados permitem operações de acesso e de modificação; operações de acesso apenas consultam um ou mais campos de uma ED, enquanto operações de modificação podem alterar os campos da estrutura.

Em geral, operações só podem ser feitas na configuração atual da ED, ou seja, ao realizar uma operação de modificação, perde-se informação sobre o "passado" da estrutura. Dizemos que, ao realizar uma operação de modificação, criamos uma nova versão da ED. Estruturas de dados *persistentes* [5] permitem realizar operações em versões criadas anteriormente. Dizemos que uma ED é *parcialmente persistente* se permite apenas operações de acesso a versões anteriores e modificação apenas na versão mais nova, e *totalmente persistente*, ou apenas *persistente*, se também permite operações de modificação em todas as versões.

Considerando um digrafo das versões onde se a versão j foi criada a partir da versão i então existe um arco de i para j, temos que para estruturas parcialmente persistentes esse digrafo é um caminho, e para estruturas totalmente persistentes é uma árvore enraizada, em que as arestas vão para longe da raiz. Este digrafo é chamado de árvore de versões.

O estudo de estruturas persistentes segue dois caminhos: técnicas gerais, para tornar *qualquer* estrutura de dados persistente, ou técnicas para tornar alguma ED específica (como uma pilha ou fila) persistente, deixando-a tão eficiente e simples quanto possível.

Persistência foi formalmente introduzida por Driscoll, Sleator e Tarjan [5], porém já era estudada anteriormente, principalmente para a implementação de estruturas de dados em linguagens funcionais, como pilhas [10], filas [6] e árvores de busca binária (ABBs [9]).

A Parte I desta dissertação detalha a solução para os problemas de ancestral comum mais profundo e ancestral de nível, usadas como caixa preta nos Capítulos 3 e 4. Pode ser pulada se já houver o conhecimento destes problemas.

A Parte II detalha a teoria de estruturas persistentes, e se divide da seguinte forma:

- Os Capítulos 3 a 6 apresentam técnicas para tornar persistentes estruturas específicas, como pilhas, filas e deques.
- O Capítulo 7 apresenta técnicas gerais para tornar persistentes certas classes de estruturas de dados. O Capítulo 8 aplica uma destas técnicas na árvore rubro-negra.
- O Capítulo 9 apresenta uma aplicação da estrutura apresentada no Capítulo 8 para resolver o problema de localização de ponto.

Parte I

Preliminares

Capítulo 1

Ancestrais em árvores

1.1 Introdução

Seja $T \coloneqq (V, E)$ um grafo. Dizemos que T é uma árvore se é acíclico e conexo. Nesse caso, entre cada par de vértices de T existe exatamente um caminho. Uma árvore é enraizada quando fixamos algum vértice $r \in V$, chamado de raiz. Para propósitos de implementação e sem perda de generalidade, supomos que $V = \{1, 2, \ldots, |V|\}$.

Para cada vértice $u \in V$, os *ancestrais* de u são os vértices no (único) caminho de u até r. O $(i-1)$-ésimo ancestral de u é o i-ésimo vértice nesse caminho, ou seja, u é o 0-ésimo ancestral de u. Dizemos que o primeiro ancestral de u, se $u \neq r$, é o *pai* de u. Defina a função $\text{Parent} : V \to V$ tal que $\text{Parent}(u)$ seja o pai do vértice u, para todo $u \in V \setminus \{r\}$, para o vértice r, pode-se considerar que $\text{Parent}(r) = r$. A *profundidade* de um vértice u é $\text{D}(u)$, o número de arestas no caminho de u até r.

O problema do Ancestral de Nível (no inglês, Level Ancestor, abreviado LA), é o problema de encontrar, dados $u \in V$ e $k \in \mathbb{Z}$ tal que $0 \leq k \leq \text{D}(u)$, o k-ésimo ancestral de u, ou seja, o problema de avaliar a função

$$\text{LA}(k, u) \coloneqq \text{Parent}^k(u).$$

O problema do Ancestral Comum de Maior Profundidade (no inglês, Lowest Common Ancestor, abreviado LCA), é o problema de encontrar, dados $u, v \in V$, o vértice w de maior profundidade que é ancestral de ambos u e v, ou seja, o problema de avaliar a função

$$\text{LCA}(u, v) \coloneqq \text{argmax}\{\text{D}(w) : w \in V,\ w \text{ é ancestral de } u \text{ e } v\}.$$

1.2 Potências de funções

Vamos considerar o problema, um pouco mais genérico, de, dada uma função $f : [n] \to [n]$ (que pode ser dada por um vetor de tamanho n, por exemplo), construir um algoritmo que, após possivelmente algum pré-processamento sobre o vetor f, responda consultas do tipo: dados $i \in [n]$ e $k \in [m] \cup \{0\}$, determinar $f^k(i)$ de forma eficiente. Se o tempo de processamento é $\mathcal{O}(p(n, m))$ e o tempo para responder cada consulta é $\mathcal{O}(q(n, m))$, dizemos que a complexidade da solução é $\langle \mathcal{O}(p(n, m)), \mathcal{O}(q(n, m)) \rangle$.

Soluções simples

Uma solução simples é não fazer nenhum pré-processamento e sempre realizar as k iterações para determinar $f^k(i)$. Esta solução tem complexidade $\langle \mathcal{O}(1), \mathcal{O}(m) \rangle$. Outra solução simples é armazenar a resposta para todas as consultas possíveis em uma matriz M tal que $M[k][i] = f^k(i)$ para todo $i \in [n]$ e $k \in [m] \cup \{0\}$. Esta matriz pode ser preenchida usando programação dinâmica em tempo $\mathcal{O}(nm)$, já que sabemos que, se $k > 0$, então $f^k(i) = f(f^{k-1}(i))$, ou seja,

$$M[k][i] = \begin{cases} f(M[k-1][i]) & \text{se } k > 0 \\ i & \text{se } k = 0, \end{cases}$$

ou seja, podemos preencher M iterando pelos seus índices em ordem não decrescente de k. Esta solução tem complexidade $\langle \mathcal{O}(nm), \mathcal{O}(1) \rangle$.

Potências de dois

As soluções apresentadas funcionam da seguinte maneira: escolhe-se uma base $(a_1, \ldots, a_x)$ tal que todo número entre 0 e m pode ser escrito como soma de zero ou mais destes números, e calcula-se (durante o pré-processamento) $f^{a_j}(i)$ para todo $i \in [n]$ e $j \in [x]$. Dado um número k, escreve-se este como $k = a_{b_1} + a_{b_2} + \ldots + a_{b_y}$, e após isso calcula-se

$$f^k(i) = f^{a_{b_1}}(f^{a_{b_2}}(\ldots(f^{a_{b_y}}(i))\ldots)).$$

Os exemplos eram simples, e no primeiro exemplos escolhemos como base apenas (1), e o tempo de consulta foi grande, enquanto no segundo exemplo escolhemos $(1, \ldots, m)$, e o tempo de pré-processamento foi grande.

Escolhendo a base de forma mais inteligente, é possível melhorar a complexidade. Cada número tem uma decomposição (única) em somas de potências de dois distintas (correspondente a sua representação binária), e existem apenas $\lfloor \lg m \rfloor$ potências de 2 entre 1 e m. Portanto, podemos escolher a base $(1, 2, 4, \ldots, 2^{\lfloor \lg m \rfloor})$. Para fazer o pré-processamento, utilizaremos programação dinâmica preenchendo uma matriz M tal que $M[k][i] \coloneqq f^{2^k}(i)$ para todo $i \in [n]$ e $0 \leq k \leq \lfloor \lg m \rfloor$. Sabemos que $f^x(f^x(i)) = f^{2x}(i)$, logo temos

$$M[k][i] = \begin{cases} M[k-1][M[k-1][i]] & \text{se } k > 0 \\ f(i) & \text{se } k = 0, \end{cases}$$

e também podemos preencher M iterando pelos seus índices em ordem não decrescente de k.

Teorema 1.1. *Todo número positivo tem uma única decomposição em soma de potências de dois distintas.*

Demonstração. A prova é o por indução. É óbvio que 1 tem uma única decomposição em soma de potências de dois.

Seja $k > 1$ um inteiro e seja 2^x a maior potência de dois menor ou igual a k. Note que $k - 2^x < 2^x$ (caso contrário $2^{x+1} \leq k$), então pela hipótese de indução $k - 2^x$ tem uma decomposição em potências de dois distintas, e nenhuma destas potências é 2^x, logo podemos adicionar esta potência

à representação. Isto prova a existência. A unicidade segue de que a representação de $k-2^x$ é única pela hipótese de indução e, como $\sum_{0\leq y<x} 2^y = 2^x - 1 < 2^x$, temos que qualquer decomposição de k deve conter 2^x. □

A prova acima nos dá um algoritmo para encontrar uma decomposição em potências de dois de k, basta encontrar a maior potência de dois menor ou igual a k e subtraí-la de k, e repetir este algoritmo até k se tornar 0.

Código 1.1: Solução para potência de função.

```
    ▷ Cria a matriz M a partir de f.
 1: function PREPROCESSING(f, n, m)
 2:     for i = 1 to n :
 3:         M[0][i] = f(i)
 4:     for k = 1 to ⌊lg m⌋ :
 5:         for i = 1 to n :
 6:             M[k][i] = M[k − 1][M[k − 1][i]]
    ▷ Devolve f^k(i), deve ser chamada após PREPROCESSING.
 7: function QUERY(k, i)
 8:     for x = ⌊lg m⌋ down to 0 :
 9:         if 2^x ≤ k :
10:             k = k − 2^x
11:             i = M[x][i]
12:     return i
```

O Código 1.1 mostra a solução discutida, que tem complexidade $\langle \mathcal{O}(n \lg m), \mathcal{O}(\lg m)\rangle$ e consome espaço $\mathcal{O}(n \lg m)$. Note que é fácil implementar QUERY(k, i) de forma que esta consuma tempo $\mathcal{O}(\lg k)$, basta, no laço, o valor de x começar com $\lfloor \lg k \rfloor$.

1.3 Ancestral de nível online para árvores

Como Parent é uma função de V em V, é possível utilizar os algoritmos discutidos na Seção 1.2 para avaliar a função LA. Note que, para cada $u \in V$ precisamos apenas calcular Parent$^k(u)$ para $k < \mathrm{D}(u)$, logo usamos $n = m = |V|$. Estes algoritmos, porém, assumem que a função é conhecida de antemão, para realizar o pré-processamento, ou seja, é necessário termos a árvore inteira para calcular a função Parent.

Na Subseção 1.2, usamos que $M[k][u] = M[k-1][M[k-1][u]]$ pode ser calculada usando programação dinâmica pois depende de índices com k menor. No caso de árvores, porém, sabemos que $f^{2^{k-1}}(u)$ é um ancestral de u, logo para calcular os valores $M[k][u]$ para todo $0 \leq k < \lfloor \lg \mathrm{D}(u) \rfloor$ basta que os valores de M já estejam calculados para todos os ancestrais de u, e então calculamos o $M[k][u]$ em ordem crescente de k (já que $M[k][u]$ depende de $M[k-1][u]$ além de seus ancestrais).

Por esse motivo, o algoritmo pode ser implementado de forma online, na qual novas folhas podem ser adicionadas à árvore. Para melhorar a notação neste caso, usamos cada nó tem um vetor *jump* com os valores de M, ou seja, $u.jump[k] = M[k][u]$. Assim que uma folha u for adicionada, podemos calcular os valores de $u.jump$, em tempo e espaço $\mathcal{O}(\lfloor \lg \mathrm{D}(u) \rfloor)$. A consulta continua da mesma forma.

Código 1.2: Solução para o problema do Ancestral de Nível.

```
    ▷ Adiciona a folha u à árvore.
1:  function AddLeaf(u)
2:      u.jump[0] = Parent(u)
3:      for k = 1 to ⌊lg D(u)⌋ :
4:          u.jump[k] = u.jump[k − 1].jump[k − 1]
    ▷ Devolve LA(k, u), deve ser chamada após AddLeaf(u).
5:  function LevelAncestor(k, u)
6:      for x = ⌊lg k⌋ down to 0 :
7:          if 2^x ≤ k :
8:              k = k − 2^x
9:              u = u.jump[x]
10:     return u
```

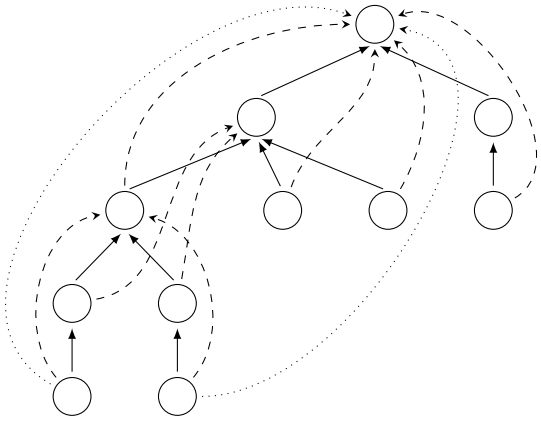

Figura 1.1: Exemplo de árvore com jump pointers. As arestas cheias são as arestas de pai (*jump*[0]), as arestas tracejadas pulam dois níveis (*jump*[1]), e as arestas pontilhadas pulam quatro níveis (*jump*[2]).

O Código 1.2 resolve o problema do Ancestral de Nível online, quando se pode adicionar folhas à árvore. A operação AddLeaf(u) consome tempo $\mathcal{O}(\lg \mathrm{D}(u))$ e a operação LevelAncestor(k, u) consome tempo $\mathcal{O}(\lg k) = \mathcal{O}(\lg \mathrm{D}(u))$. Note que cada nó u deve armazenar um vetor de tamanho $\lfloor \lg \mathrm{D}(u) \rfloor$. Na literatura, essa técnica é chamada de Sparse Table, já que "armazenamos" uma tabela de tamanho $|V| \times |V|$ usando apenas espaço $|V| \lg |V|$, ou Jump Pointers, neste caso mais específico do Ancestral de Nível em árvores [1]. A Figura 1.1 ilustra os jump pointers de uma árvore.

1.4 Cálculo do ancestral comum mais profundo

Seja $u, v \in V$ e c o ancestral comum mais profundo de u e v, assuma, sem perda de generalidade que $\mathrm{D}(v) \geq \mathrm{D}(u)$. É óbvio que $\mathrm{D}(c) \leq \mathrm{D}(u)$, logo podemos "nivelar" estes vértices, ou seja, trocar v por $\mathrm{LA}(\mathrm{D}(v) - \mathrm{D}(u), v)$.

Assim, temos que $\mathrm{D}(u) = \mathrm{D}(v)$, e, para encontrar o ancestral comum mais profundo de u e v, devemos encontrar o menor $k^\star$ tal que $\mathrm{LA}(k^\star, u) = \mathrm{LA}(k^\star, v)$. Note que esta função é monótona

em k, ou seja, se $k < \mathrm{D}(u) - 1$ e $\mathrm{LA}(k, u) = \mathrm{LA}(k, v)$, então

$$\mathrm{LA}(k+1, u) = \mathrm{Parent}(\mathrm{LA}(k, u)) = \mathrm{Parent}(c) = \mathrm{Parent}(\mathrm{LA}(k, v)) = \mathrm{LA}(k+1, v).$$

Isso permitirá a utilização dos Jump Pointers para determinar tal $k^\star$ mínimo.

Se $x < k^\star$ então $\mathrm{LA}(x, u) \neq \mathrm{LA}(x, v)$, ou seja, temos uma forma de checar se $x < k^\star$ sem conhecer $k^\star$. Como, pela prova do Teorema 1.1, para decompor um número em soma de potências de dois distintas basta determinar a maior potência de dois menor ou igual a esse número, adicioná-la à decomposição e repetir o algoritmo, e temos jump pointers para as potências de dois para todos os nós, conseguimos determinar a decomposição em potências de dois distintas de $k^\star - 1$ (já que conseguimos testar $x \leq k^\star - 1$ e não $x \leq k^\star$), e de forma similar à função LevelAncestor do Código 1.2, podemos determinar o $(k^\star - 1)$-ésimo ancestral de u e v em tempo $\mathcal{O}(\lg \mathrm{D}(u))$.

Código 1.3: Ancestral Comum Mais Profundo usando Jump Pointers.

```
1:  function LowestCommonAncestor(u, v)
2:      if D(u) > D(v) :
3:          u, v = v, u                          ▷ Garantindo que D(u) ≤ D(v).
4:      v = LevelAncestor(D(v) − D(u), v)        ▷ Nivelando v.
5:      if u = v :
6:          return u
7:      for i = ⌊lg D(u)⌋ down to 0 :
8:          if u.jump[i] ≠ v.jump[i] :
9:              u = u.jump[i]
10:             v = v.jump[i]
        ▷ u é agora o filho do LCA de u e v.
11:     return Parent(u)
```

Invariante. *Ao início da iteração com valor i do* ***for*** *da linha 7 do Código 1.3, vale que $0 < \mathrm{D}(u) - \mathrm{D}(c) \leq 2^{i+1}$, onde $c = \mathrm{LCA}(u, v)$, e o valor de $\mathrm{LCA}(u, v)$ não se altera ao final da iteração.*

Demonstração. Para a base, $i = \lfloor \lg \mathrm{D}(u) \rfloor$ e

$$\mathrm{D}(u) - \mathrm{D}(c) \leq \mathrm{D}(u) = 2^{\lg \mathrm{D}(u)} \leq 2^{\lfloor \lg \mathrm{D}(u) \rfloor + 1},$$

além disso, se $u = c$, então o **if** da linha 5 é executado, logo $\mathrm{D}(u) - \mathrm{D}(c) > 0$.

Suponha que o invariante vale ao início da iteração com valor i, e provaremos que continua valendo ao começo da iteração com valor $i-1$. Seja $d \coloneqq \mathrm{D}(u) - \mathrm{D}(c)$, sabemos então que $0 < d \leq 2^{i+1}$. Note que $d \leq 2^i \iff \mathrm{LA}(2^i, u) = \mathrm{LA}(2^i, v)$, ou seja, o **if** da linha 8 é executado se e somente se $d > 2^i$. Então

- se $d \leq 2^i$, o invariante já vale para $i - 1$ e o **if** não é executado;
- se $d > 2^i$, o **if** é executado, portanto trocamos u por $\mathrm{LA}(2^i, u)$ e v por $\mathrm{LA}(2^i, v)$. Seja d' o novo valor de $\mathrm{D}(u) - \mathrm{D}(c)$. A profundidade de u e v diminui de 2^i, logo o valor de

$$d' = d - 2^i \leq 2^{i+1} - 2^i = 2^i.$$

	Tempo/Espaço
AddLeaf(u)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
LevelAncestor(k, u)	$\mathcal{O}(\lg n)$
LowestCommonAncestor(u, v)	$\mathcal{O}(\lg n)$

Tabela 1.1: Consumo de tempo e espaço da solução discutida, onde n é o tamanho da árvore.

Como $d > 2^i$, também temos que $d' = d - 2^i > 0$, e a invariante continua a valer.

□

Portanto, ao final da última iteração, a invariante vale para $i = -1$, ou seja, vale que $0 < \mathrm{D}(u) - \mathrm{D}(c) \leq 2^0 = 1$, logo c é o pai de u e a função retorna o valor correto.

A Tabela 1.1 mostra o consumo de tempo e espaço das implementações discutidas nesse capítulo.

Capítulo 2

LA com representação skew binary

Neste capítulo, apresentamos uma outra solução para o problema do Ancestral de Nível. Esta solução, apesar de um pouco mais complicada, requer processamento que consome espaço e tempo constante por nó adicionado. Está solução foi dada inicialmente por Myers [10].

2.1 Definição e propriedades

Um número *skew-binary* de tamanho n é uma string $a = a_n a_{n-1} \dots a_1$ tal que $a_i \in \{0, 1, 2\}$ e $a_n \neq 0$. O valor de tal número é $\mathrm{V}(a) \coloneqq \sum_{i=1}^{n} a_i(2^i - 1)$. Note que múltiplas strings podem ter o mesmo valor, por exemplo, ambos 21 e 100 têm valor 7.

Para todo a tal que $\mathrm{V}(a) \neq 0$, defina $\mathrm{NZ}(a) \coloneqq \min\{i \in [n] : a_i \neq 0\}$, ou seja, a posição do dígito não nulo menos significativo de a. Dizemos que um número skew-binary é *canônico* se todos os seus dígitos são 0 ou 1 exceto, possivelmente, o dígito não nulo menos significativo. Mais formalmente, se $a_i = 2 \implies i = \mathrm{NZ}(a)$. Seja *CSB* o conjunto de todos os números skew-binary.

Esta igualdade será útil na prova dos lemas:

$$\sum_{i=l}^{r} 2^i = 2^{r+1} - 2^l. \tag{A}$$

Lema 2.1. *Se $a \in CSB$ e $|a| = n$, então $2^n - 1 \leq \mathrm{V}(a) \leq 2^{n+1} - 2$.*

Demonstração. Considere o número $b = 10^{n-1}$. Como, por definição, $a_n \geq 1 = b_n$ e $a_i \geq 0 = b_i$ para $i \in [n-1]$, vale que $V(b) \geq V(a) = 2^n - 1$.

Note que

$$\begin{aligned} V(a) = \sum_{i=\mathrm{NZ}(a)}^{n} a_i(2^i - 1) &\overset{(1)}{\leq} \left(\sum_{i=\mathrm{NZ}(a)+1}^{n} (2^i - 1) \right) + 2(2^{\mathrm{NZ}(a)} - 1) \\ &\overset{(2)}{=} 2^{n+1} - 2^{\mathrm{NZ}(a)+1} - (n - \mathrm{NZ}(a)) + 2^{\mathrm{NZ}(a)+1} - 2 \\ &\overset{(3)}{\leq} 2^{n+1} - 2, \end{aligned}$$

onde (1) vale por que $a \in CSB$, (2) vale por (A) e (3) vale pois $\mathrm{NZ}(a) \leq n$.

□

O lema mostra que o menor e maior número de n dígitos em CSB são 10^{n-1} e 20^{n-1}, respectivamente, e também que qualquer representação de x em CSB usa $\lfloor \lg(x+1) \rfloor$ dígitos.

Teorema 2.2. *Cada número tem uma representação única em CSB. Equivalentemente,* $\mathrm{V} : \mathit{CSB} \to \mathbb{N}$ *é uma função bijetora.*

Demonstração. Vamos provar que V é injetora, ou seja, se $a \neq b$ então $\mathrm{V}(a) \neq \mathrm{V}(b)$. Suponha, sem perda de generalidade, que $|a| \geq |b|$. Se $|a| > |b|$, então, pelo Lema 2.1,

$$\mathrm{V}(a) \geq 2^{|a|} - 1 > 2^{|a|} - 2 \geq 2^{|b|+1} - 2 \geq \mathrm{V}(b).$$

Se $|a| = |b|$, considere $i^\star = \max\{i \in [n] : a_i \neq b_i\}$. Assuma, sem perda de generalidade, que $a_{i^\star} > b_{i^\star}$. Escreva $a = \alpha a_{i^\star} \beta$ e $b = \alpha b_{i^\star} \gamma$. Então

$$V(a) - V(b) \overset{(1)}{\geq} (2^{i^\star} - 1) + V(\beta) - V(\gamma) \overset{(2)}{\geq} (2^{i^\star} - 1) - (2^{i^\star} - 2) = 1,$$

onde (1) vale pois $a_i \geq b_i + 1$, (2) vale pois $\mathrm{V}(\beta) \geq 0$ e usando o Lema 2.1 sobre γ. Isso prova que $\mathrm{V}(a) \neq \mathrm{V}(b)$, logo V é injetora.

Para provar que V é sobrejetora, precisamos provar que, para todo $x \in \mathbb{N}$, existe $a \in \mathit{CSB}$ tal que $V(a) = x$. Por indução em n, vamos provar que, para todo $x \leq 2^{n+1} - 2$ existe $a \in \mathit{CSB}$ tal que $V(a) = x$. Se $n = 0$, então $a = 0$ é tal que $V(a) = 0$. Suponha que a hipótese vale para todo $x \leq 2^n - 2$. Seja $2^n - 1 \leq y \leq 2^{n+1} - 2$. Se $y = 2^{n+1} - 2$, sabemos que $a = 20^{n-1} \in \mathit{CSB}$ é tal que $V(a) = y$. Caso contrário,

$$y - (2^n - 1) < 2^{n+1} - 2 - (2^n - 1) = 2^n - 1,$$

logo existe $a \in \mathit{CSB}$ tal que $V(a) = y - (2^n - 1)$, mas então $b = 1a \in \mathit{CSB}$ e $V(b) = y$. □

Como cada número tem uma representação única em CSB, podemos definir uma função $\mathrm{R} : \mathbb{N} \to \mathit{CSB}$ tal que $V(\mathrm{R}(x)) = x$ para todo $x \in \mathbb{N}$.

Lema 2.3. *Seja* $a \in \mathit{CSB}$ *tal que* $\mathrm{V}(a) > 0$. *Se* $\mathrm{NZ}(a) = 1$ *então* $V(a_n \dots a_2(a_1 - 1)) = V(a) - 1$, *caso contrário* $V(a_n \dots a_{\mathrm{NZ}(a)+1}(a_{\mathrm{NZ}(a)} - 1)20^{\mathrm{NZ}(a)-2}) = V(a) - 1$.

Demonstração. Quando $\mathrm{NZ}(a) = 1$, vale que $a_1 \neq 0$, logo $b \coloneqq a_n \dots a_2(a_1 - 1) \in \mathit{CSB}$. Além disso, $V(a) - V(b) = a_1 - b_1 = 1$.

Caso contrário, $a_{\mathrm{NZ}(a)} \neq 0$, e como o único dígito em a que pode ser 2 é o $\mathrm{NZ}(a)$-ésimo dígito, temos que $b \coloneqq a_n \dots a_{\mathrm{NZ}(a)+1}(a_{\mathrm{NZ}(a)} - 1)20^{\mathrm{NZ}(a)-2} \in \mathit{CSB}$. Além disso,

$$V(a) - V(b) = (a_{\mathrm{NZ}(a)} - b_{\mathrm{NZ}(a)})(2^{\mathrm{NZ}(a)} - 1) - 2(2^{\mathrm{NZ}(a)-1} - 1) = 2^{\mathrm{NZ}(a)} - 1 - (2^{\mathrm{NZ}(a)} - 2) = 1.$$

□

O lema mostra que subtração por 1 em skew-binary canônico consiste de diminuir em 1 o dígito não nulo menos significativo e, se existir, aumentar para 2 o dígito à direita deste.

2.2 Jump pointers

No problema do Ancestral de Nível, para avaliar $\mathrm{LA}(k, u)$, temos um nó u de profundidade $\mathrm{D}(u)$ e queremos determinar seu ancestral v de profundidade $\mathrm{D}(v) = \mathrm{D}(u) - k$. Na solução apresentada na Subseção 1.2, cada nó tinha um ponteiro para seu 2^x-ancestral, para todo $x \in \lfloor \lg \mathrm{D}(u) \rfloor$, e o nó v era alcançado a partir de u pulando as potências de dois distintas da decomposição de k.

Este problema pode ser interpretado de outra forma. Temos um número x e queremos transformá-lo em $y \leq x$, a cada passo diminuindo o valor de x. Com esta interpretação, na solução da Subseção 1.2, que tenta transformar $\mathrm{D}(u)$ em $\mathrm{D}(v)$, a partir de cada número podíamos subtrair qualquer potência de dois. Note que diminuir o valor de um número por z é equivalente a escolher o z-ésimo ancestral de um nó.

Vamos considerar uma solução alternativa para este problema, na qual partir de um número x positivo podemos pular para os números $x - 1$ ou $\mathrm{J}(x)$, onde $\mathrm{J}(x) \coloneqq x - (2^{\mathrm{NZ}(\mathrm{R}(x))} - 1)$, ou seja, se considerarmos $\mathrm{R}(x)$, a representação em skew-binary canônico de x, $\mathrm{J}(x)$ consiste de diminuir em um o dígito não nulo menos significativo de $\mathrm{R}(x)$. Por exemplo, se $x = 13$, então $\mathrm{R}(x) = 120$ e $\mathrm{J}(x) = \mathrm{V}(110) = 10$.

Considere o seguinte algoritmo, que transforma x em y (inicialmente $x > y$):

Código 2.1: Transformando x em y usando $x - 1$ e $\mathrm{J}(x)$.

```
1: while x ≠ y :
2:     if J(x) ≥ y :
3:         x = J(x)
4:     else
5:         x = x − 1
```

O algoritmo é guloso no sentido que sempre escolhe usar J quando possível (e $\mathrm{J}(x) \leq x - 1$). A correção do algoritmo é clara, já que é sempre possível alcançar y usando apenas $x - 1$, e x nunca se torna menor que y.

Teorema 2.4. *O algoritmo no Código 2.1 termina em $\mathcal{O}(\lg x)$ iterações do **while**.*

Demonstração. Seja $a \coloneqq \mathrm{R}(x)$ e $b \coloneqq \mathrm{R}(y)$ e $n \coloneqq |a|$. Se $|a| > |b|$, aumente b adicionando 0s à esquerda. Seja $i^\star = \max\{i \in [n] : a_i \neq b_i\}$, e escreva $a = \alpha a_{i^\star} \beta$ e $b = \alpha b_{i^\star} \gamma$. Pelo Lema 2.1 (e também usado na prova do Teorema 2.2), temos que $a_{i^\star} > b_{i^\star}$ e que $V(c) > V(b)$, onde $c \coloneqq b_n \dots b_{i^\star+1}(b_{i^\star} + 1)0^{i^\star-1}$. Note que $c_i \leq a_i$ para todo $i \in [n]$, e temos que $\mathrm{NZ}(a) \leq i^\star$. Se $\mathrm{NZ}(a) < i^\star$ vale que $a_{\mathrm{NZ}(a)} > 0 = c_{\mathrm{NZ}(a)}$, e se $\mathrm{NZ}(a) = i^\star$ vale que $a_{\mathrm{NZ}(a)} = a_{i^\star} \geq b_{i^\star} + 1 = c_{i^\star}$. Portanto $a_{\mathrm{NZ}(a)} \geq c_{\mathrm{NZ}(a)}$, com igualdade apenas se $a = c$, logo se $a \neq c$ podemos diminuir $a_{\mathrm{NZ}(a)}$ em um, ou seja, $J(V(a)) \geq V(c)$. Podemos repetir este argumento até que $a = c$, ou seja $x = V(c)$. Esta parte realiza no máximo $\sum_{i=1}^{i^\star} a_i \leq i^\star + 1$ iterações, já que cada dígito é no máximo 1, a menos de um destes, que pode ser 2.

Vamos provar que, se $a = b_n \dots b_{i+1}(b_i + 1)0^{i-1}$ para algum $1 \leq i \leq n$, com $x = \mathrm{V}(a)$ e $y = \mathrm{V}(b)$, então o algoritmo terminará em no máximo $2i$ iterações. Com isso provado, começando de $c = b_n \dots b_{i^\star+1}(b_{i^\star} + 1)0^{i^\star-1}$, o algoritmo realiza no máximo $2i^\star$ iterações adicionais, logo o algoritmo realiza no máximo $3i^\star + 1 \leq 3n + 1 = \mathcal{O}(\lg x)$ iterações no total, e o teorema estará provado.

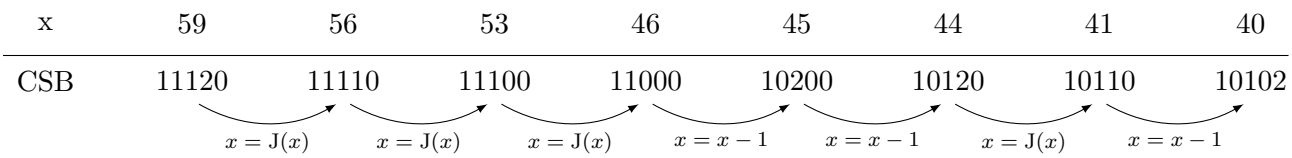

Figura 2.1: Exemplo do algoritmo do Código 2.1 aplicado sobre $x = 59$ e $y = 40$.

A prova será por indução em i. Se $i = 1$ então $J(x) = x - 1 = y$ e o algoritmo termina em uma iteração.

Suponha que a hipótese vale para valores menores que i. Se $\mathrm{NZ}(b) \geq i$, ou seja, b tem um sufixo de pelo menos $i - 1$ valores 0, ou seja $a = b_n \dots b_{i+1}(b_i + 1)b_{i-1} \dots b_1$, e então se diminuirmos 1 de a_i temos b, ou seja, $\mathrm{J}(\mathrm{V}(a)) = \mathrm{V}(b)$, e o algoritmo termina em uma iteração. Caso contrário, $\mathrm{J}(\mathrm{V}(a)) < \mathrm{V}(b)$, e então $x = x - 1$ é executado. Pelo Lema 2.3, $\mathrm{R}(x - 1) = b_n \dots b_i 20^{i-2}$, e é claro que $\mathrm{V}(20^{i-2}) \geq \mathrm{V}(b_{i-1} \dots b_1)$. Considere o valor de b_{i-1}:

- se $b_{i-1} = 2$, então $x - 1 = y$ e o algoritmo termina, gastando uma iteração;
- se $b_{i-1} = 1$, então podemos aplicar a hipótese de indução (para $i - 1$) e temos que o algoritmo gasta no máximo $2(i - 1) + 1 < 2i$ iterações;
- se $b_{i-1} = 0$, então $\mathrm{J}(x - 1) > y$ e $\mathrm{R}(\mathrm{J}(x - 1)) = b_n \dots b_i 10^{i-2}$, logo podemos aplicar a hipótese de indução (para $i - 1$) e temos que o algoritmo gasta no máximo $2(i - 1) + 2 = 2i$ iterações.

□

A Figura 2.1 mostra um exemplo de aplicação do algoritmo no Código 2.1. A transformação de $x = 59$ até $x = 46$ corresponde à primeira parte da prova do Teorema 2.4, e o resto corresponde à segunda.

Portanto, é possível usar a mesma ideia para resolver o problema do Ancestral de Nível. Suponha que todo vértice u tenha um campo $u.jump$ que armazene seu ancestral com profundidade $\mathrm{J}(\mathrm{D}(u))$. Então, de forma análoga ao algoritmo do Código 2.1, podemos resolver o problema do Ancestral de Nível como no Código 2.2. A correção é clara e o consumo de tempo $\mathcal{O}(\lg \mathrm{D}(u))$ segue diretamente do Teorema 2.4.

Código 2.2: Algoritmo para Ancestral de Nível usando a representação skew-binary.

```
1: function LevelAncestor(k, u)
2:     y = D(u) − k
3:     while D(u) ≠ y :
4:         if D(u.jump) ≥ y :
5:             u = u.jump
6:         else
7:             u = Parent(u)
8:     return u
```

2.3 Computando jump pointers

A seção anterior apresentou uma solução para o problema do Ancestral de Nível que consome tempo logarítmico, porém, assumimos que cada nó u tinha um campo $u.jump$ com seu $\mathrm{J}(\mathrm{D}(u))$-ésimo ancestral. Nesta seção, detalharemos como encontrar tal ancestral para cada nó.

Teorema 2.5. *Seja $a \in CSB$. Se a não contém nenhum dígito dois, ou seja, se $a_{\mathrm{NZ}(a)} \neq 2$, então $\mathrm{J}(\mathrm{V}(a)+1) = \mathrm{V}(a)$. Caso contrário, $\mathrm{J}(\mathrm{V}(a)+1) = \mathrm{J}(\mathrm{J}(\mathrm{V}(a))$.*

Demonstração. Suponha que $a_{\mathrm{NZ}(a)} \neq 2$. Note que $b \coloneqq a_n \dots a_2(a_1+1)$ é um número em skew-binary tal que $\mathrm{V}(b) = \mathrm{V}(a)+1$, e, já que a não tem dígito 2, vale que $b \in CSB$. Obviamente $\mathrm{NZ}(b) = 1$, logo $\mathrm{J}(\mathrm{V}(a)+1) = \mathrm{J}(\mathrm{V}(b)) = \mathrm{V}(b) - 1 = \mathrm{V}(a)$.

Caso contrário, $a_{\mathrm{NZ}(a)} = 2$. Considere $b \coloneqq a_n \dots a_{\mathrm{NZ}(a)+2}(a_{\mathrm{NZ}(a)+1}+1)0^{\mathrm{NZ}(a)}$, pelo Lema 2.3 vale que $R(V(b)-1) = a$, logo $V(b) = V(a)+1$. Note que

$$\begin{aligned} \mathrm{J}(\mathrm{V}(b)) &= a_n \dots a_{\mathrm{NZ}(a)+1}0^{\mathrm{NZ}(a)}, \\ \mathrm{J}(\mathrm{V}(a)) &= a_n \dots a_{\mathrm{NZ}(a)+1}10^{\mathrm{NZ}(a)-1}, \text{ e} \\ \mathrm{J}(\mathrm{J}(\mathrm{V}(a))) &= a_n \dots a_{\mathrm{NZ}(a)+1}0^{\mathrm{NZ}(a)}. \end{aligned}$$

Portanto, $\mathrm{J}(V(a)+1) = \mathrm{J}(\mathrm{J}(\mathrm{V}(a)))$. □

Pelo Teorema 2.5, é fácil calcular $\mathrm{J}(x)$ a partir dos valores J calculados para valores menores que x, se soubermos identificar se $\mathrm{R}(x-1)$ tem algum dígito 2.

Proposição 2.6. *Seja $a \in CSB$ tal que $\mathrm{V}(a) \neq 0$. Então*

$$a_{\mathrm{NZ}(a)} = 2 \iff \mathrm{J}(\mathrm{V}(a)) \neq 0 \text{ e } \mathrm{V}(a) - \mathrm{J}(\mathrm{V}(a)) = \mathrm{J}(\mathrm{V}(a)) - \mathrm{J}(\mathrm{J}(\mathrm{V}(a))).$$

Demonstração. Seja $b \coloneqq \mathrm{R}(\mathrm{J}(\mathrm{V}(a)))$ e $c \coloneqq \mathrm{R}(\mathrm{J}(\mathrm{V}(b)))$. Lembre que $\mathrm{J}(\mathrm{V}(a)) = \mathrm{V}(a) - (2^{\mathrm{NZ}(a)} - 1)$.

Se $a_{\mathrm{NZ}(a)} = 2$, então $\mathrm{NZ}(b) = \mathrm{NZ}(a)$ (e $\mathrm{V}(b) \neq 0$), logo segue que

$$\mathrm{V}(a) - \mathrm{J}(\mathrm{V}(a)) = 2^{\mathrm{NZ}(a)} - 1 = 2^{\mathrm{NZ}(b)} - 1 = \mathrm{V}(b) - \mathrm{J}(\mathrm{V}(b)) = \mathrm{J}(\mathrm{V}(a)) - \mathrm{J}(\mathrm{J}(\mathrm{V}(a))).$$

Já se $a_{\mathrm{NZ}(a)} = 1$, então se $\mathrm{J}(\mathrm{V}(a)) \neq 0$ vale que $\mathrm{NZ}(b) > \mathrm{NZ}(a)$, logo segue que

$$\mathrm{V}(a) - \mathrm{J}(\mathrm{V}(a)) = 2^{\mathrm{NZ}(a)} - 1 < 2^{\mathrm{NZ}(b)} - 1 = \mathrm{V}(b) - \mathrm{J}(\mathrm{V}(b)) = \mathrm{J}(\mathrm{V}(a)) - \mathrm{J}(\mathrm{J}(\mathrm{V}(a))).$$

□

A Proposição 2.6 nos dá uma maneira de verificar se a representação skew-binary de x tem algum dígito 2, se já tivermos calculado os valores de J para números menores ou iguais a x.

O Código 2.3 mostra como adicionar uma folha de forma online à árvore, da mesma forma como no Código 1.2. Nesse caso, porém, o consumo de tempo e espaço é constante. A correção segue diretamente do Teorema 2.5 e da Proposição 2.6, já que a comparação da linha 3 verifica as condição dadas pela proposição e o **if** computa o campo *jump* (equivalente à função J) como no teorema. Note que a raiz r tem profundidade 0 e seu campo *jump* não é usado.

Código 2.3: Adicionando uma folha à árvore com raiz r.

```
1: function AddLeaf(u)
2:     v = Parent(u)
3:     if v.jump ≠ r and D(v) − D(v.jump) = D(v.jump) − D(v.jump.jump) :
4:         u.jump = v.jump.jump
5:     else
6:         u.jump = v
```

	Tempo/Espaço
AddLeaf(u)	$\mathcal{O}(1)/\mathcal{O}(1)$
LevelAncestor(k, u)	$\mathcal{O}(\lg n)$
LowestCommonAncestor(u, v)	$\mathcal{O}(\lg n)$

Tabela 2.1: Consumo de tempo e espaço da solução com skew-binary, onde n é o tamanho da árvore.

2.4 Ancestral comum mais profundo

Para encontra o ancestral comum mais profundo de dois vértices u e v, usamos a mesma lógica descrita na Seção 1.4. Inicialmente nivelamos u e v para terem a mesma profundidade. Seja c o ancestral comum mais profundo de u e v, note que no Código 2.1 não é realmente necessário conhecermos y, apenas precisamos determinar, a cada iteração, se $\mathrm{J}(x) \geq y$. Sabemos que $u.jump = v.jump \iff D(u.jump) > D(c)$, ou seja, podemos determinar se $J(D(u)) \geq D(c) + 1$.

Dessa forma, conseguimos encontrar o ancestral de u com profundidade $D(c) + 1$, e o pai deste será c. Veja o Código 2.4, e note sua similaridade com o Código 1.3.

Código 2.4: Ancestral Comum Mais Profundo usando representação skew-binary

```
1: function LowestCommonAncestor(u, v)
2:     if D(u) > D(v) :
3:         u, v = v, u                          ▷ Garantindo que D(u) ≤ D(v).
4:     v = LevelAncestor(D(v) − D(u), v)        ▷ Nivelando v.
5:     if u = v :
6:         return u
7:     while Parent(u) ≠ Parent(v) :
8:         if u.jump ≠ v.jump :
9:             u = u.jump
10:            v = v.jump
11:        else
12:            u = Parent(u)
13:            v = Parent(v)
       ▷ u é agora o filho do LCA de u e v.
14:    return Parent(u)
```

A Tabela 2.1 mostra o consumo de tempo e espaço das implementações discutidas nesse capítulo. A Tabela 2.2 compara o consumo de tempo das implementações de Ancestral de Nível e Ancestral Comum Mais Profundo apresentadas nesse trabalho.

	Representação binária	Skew-binary
AddLeaf(u)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
LevelAncestor(k, u)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$
LowestCommonAncestor(u, v)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 2.2: Comparação do consumo de tempo e espaço das soluções dos Capítulos 1 e 2, onde n é o tamanho da árvore.

Parte II

Persistência

Capítulo 3

Pilhas e filas de acesso aleatório

Pilhas são uma das estruturas de dados mais simples. Uma pilha é *de acesso aleatório* se permite acesso a qualquer elemento seu, não só o topo. Discutiremos neste capítulo como transformá-las em estruturas persistentes. Em nosso pseudocódigo, usaremos estruturas de dados como objetos. As operações aplicadas sobre uma ED são funções que recebem essa ED como argumento (além de possíveis argumentos adicionais). Além disso, existe uma operação que devolve uma ED vazia.

Para manipular estruturas persistentes, é conveniente que, ao chamar uma operação de modificação, uma nova versão da ED seja devolvida, e a versão anterior continue válida, ou seja, continue podendo ser usada para operações de acesso, ou de modificação. O estado da versão anterior pode mudar, mas a resposta para qualquer operação de acesso deve ser a mesma. Assim, nesse contexto, uma pilha de acesso aleatório é uma lista de elementos, que permite as seguintes operações:

- STACK()
  Devolve uma pilha vazia.
- PUSH(p, x)
  Devolve uma cópia da pilha p com o valor x inserido no seu fim.
- POP(p)
  Devolve uma cópia da pilha p com seu último elemento removido.
- SIZE(p)
  Devolve o número de elementos na pilha p.
- TOP(p)
  Devolve o último elemento da pilha p.
- K-TH(p, k)
  Devolve o k-ésimo elemento da pilha p.

As operações são o meio com o qual o "usuário" lida com a ED, no resto do capítulo descreveremos como implementar tais operações. Para deixar clara a notação, operações são SUBLINHADAS e funções auxiliares (que usamos para nos ajudar a implementar as operações) não são.

Na literatura, é usual chamar o último elemento de uma pilha de *topo*, porém, como a operação K-TH necessita que os elementos tenham uma ordem e sentido, deixamos claro na descrição das operações qual é o primeiro e o último elemento.

Temos que PUSH(p, x) e POP(p) são operações de modificação, enquanto TOP(p), K-TH(p, k) e SIZE(p) são operações de acesso.

3.1 Persistência total

Para implementar uma pilha persistente, utilizaremos a implementação de pilhas usando lista ligada. Essa implementação consiste de vários nós, cada um com três campos: *val*, com o valor armazenado neste nó, *next*, com um ponteiro para o próximo nó na lista e *size* com o tamanho da lista começando naquele nó. O último nó da lista ligada tem como seu campo *next* um valor especial **null**, que indica que este é o último nó da lista.

Uma pilha pode ser representada pelo nó correspondente ao começo de sua lista. Note que o campo *val* do primeiro nó da lista é o último elemento da pilha, e assim por diante. Considere que estamos apenas fazendo uma pilha, sem preocupação com persistência. Para realizar a operação PUSH(p, x), basta criar um novo nó com valor x e fazer seu campo *next* apontar para o começo anterior da lista. O novo nó passa a ser o novo começo da lista. Para realizar a operação TOP(p), basta devolver o campo *val* do nó do começo, e para realizar POP(p), consideramos que o novo começo é o nó apontado pelo campo *next* do começo anterior.

Perceba que nunca é necessário mudar os campos de algum nó já criado; em particular, efetivamente não se remove da lista nenhum nó. Nesse caso, se guardarmos em p_i o nó do começo da lista na i-ésima versão, os nós *acessíveis a partir de* p_i e os campos destes não mudarão mesmo com as operações realizadas em versões futuras. Portanto é possível realizar operações de acesso e modificação em versões anteriores da estrutura, e esta então é uma estrutura persistente.

Implementações de estruturas como esta, nas quais operações de modificação nunca mudam nenhum valor existente da ED, apenas criam valores novos, são chamadas de *funcionais*. Como qualquer valor acessível a partir de uma versão antiga continua o mesmo, estas estruturas são sempre persistentes. Note que, diferente de implementações usuais, não é permitido apagar valores que não serão mais usados na versão atual. Dizemos que a versão atual é a última versão criada por alguma operação de modificação.

O Código 3.1 mostra a implementação de tal pilha persistente. Usamos que **new** NODE(x, nx, sz) cria um novo nó com campos $val = x$, $next = nx$ e $size = sz$. Note que ignoramos a operação K-TH(p, k) por ora, mas voltaremos a discutir esta na Seção 3.3.

3.2 Exemplo

Vamos considerar a sequência de operações no Exemplo 3.1. Na esquerda temos as operações realizadas, e na direita as novas pilhas criadas, ou o valor devolvido pela operação TOP(p).

Em uma pilha efêmera (não persistente) adicionamos apenas elementos no começo da lista ligada, mas no caso da pilha persistente podemos adicionar nós em outros pontos, e na verdade a estrutura resultante é uma arborescência, ou seja, uma árvore enraizada onde as arestas apontam para a raiz,

Código 3.1: Pilha persistente.

```
1: function Stack()
2:     return null
3: function Size(p)
4:     if p = null :
5:         return 0
6:     else
7:         return p.size
8: function Push(p, x)
9:     return new Node(x, p, Size(p) + 1)
10: function Top(p)
11:     return p.val
12: function Pop(p)
13:     return p.next
```

$p_0 = \text{Stack}()$	p_0 :
$p_1 = \text{Push}(p_0, 5)$	p_1 : 5
$p_2 = \text{Push}(p_1, 7)$	p_2 : 5 7
$p_3 = \text{Push}(p_2, 6)$	p_3 : 5 7 6
$p_4 = \text{Pop}(p_2)$	p_4 : 5
$\text{Top}(p_3)$	Devolve 6
$p_5 = \text{Push}(p_4, 9)$	p_5 : 5 9
$\text{Top}(p_4)$	Devolve 5
$p_6 = \text{Push}(p_0, 5)$	p_6 : 5

Exemplo 3.1: Exemplo de uso de uma pilha persistente.

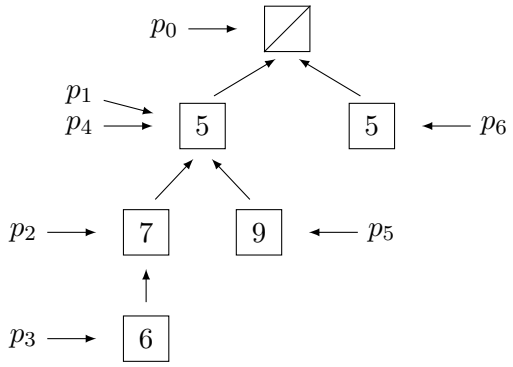

Figura 3.1: Arborescência criada pela sequência de operações do Exemplo 3.1.

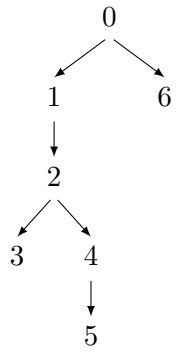

Figura 3.2: Árvore de versões associada à sequência de operações do Exemplo 3.1.

considerando que o apontador da raiz é **null**. Note que, como discutido anteriormente, a partir de cada nó podemos apenas acessar uma lista, se considerarmos seus campos *next*.

A Figura 3.1 mostra a arborescência criada para a sequência de operações do Exemplo 3.1. Em cada nó é indicado o valor armazenado naquele nó, e a flecha saindo de cada nó indica seu campo *next*. Os valores $p_0, \ldots, p_6$ apontam para os seus nós correspondentes. Note que é possível que $p_i = p_j$ para $i \neq j$. Isso ocorre no exemplo, onde $p_4 = p_1$. Isso não ocorre sempre que as pilhas têm os mesmos valores; perceba que $p_1 \neq p_6$, apesar destas duas pilhas terem os mesmos elementos na mesma ordem.

Perceba também que a árvore de versões não é igual à árvore da estrutura. A Figura 3.2 mostra a árvore de versões para essa sequência de operações. Nos nós estão os índices das versões (a i-ésima modificação cria a versão i), e a versão i é o pai da versão j se a versão j foi criada usando a versão i.

3.3 Acesso a outros elementos

Apresentaremos a implementação da função $\underline{\text{K-TH}}(p, k)$. Esta operação é uma generalização de $\underline{\text{TOP}}(p)$, já que $\underline{\text{TOP}}(p) = \underline{\text{K-TH}}(p, p.size)$.

Lembre-se que p é um nó de uma arborescência, e representa o último elemento da pilha, então $\underline{\text{K-TH}}(p, k)$ precisa determinar o $(p.size - k)$-ésimo ancestral de p, ou seja, o nó alcançado a partir de p ao caminhar por $p.size - k$ links *next*. Nesse caso, o 0-ésimo ancestral de um nó é o próprio nó.

Note que esse é o problema do Ancestral de Nível, discutido nos Capítulos 1 e 2. Nes-

	Representação Binária	Skew-Binary
Stack()	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$
Push(p, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
Pop(p)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Top(p)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Size(p)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
k-th(p, k)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 3.1: Consumo de tempo e espaço da implementação de uma pilha de acesso aleatório persistente, usando cada uma das implementações de Ancestral de Nível, onde n é o tamanho atual da pilha.

ses capítulos foi discutido como encontrar ancestrais de níveis usando as operações AddLeaf(u) e LevelAncestor(k, u). Note que apresentamos essas operações de forma que folhas novas possam ser adicionadas de forma online, que é necessário neste problema, já que adicionamos uma folha quando aplicamos uma operação de modificação à pilha.

Se assumimos que a função **new** Node(x, nx, sz) também chama a operação AddLeaf(u) para o novo nó criado, com Parent$(u) = nx$, então o Código 3.2 implementa a operação k-th(p, k), e está trivialmente correto.

Código 3.2: Implementação de k-th(p, k) usando Ancestral de Nível como caixa preta.

```
1: function k-th(p, k)
2:     return LevelAncestor(p.size − k, p).val
```

A Tabela 3.1 mostra o consumo de tempo e espaço de uma pilha persistente, usando o Ancestral de Nível do Capítulo 1 e 2.

3.4 Filas persistentes

Filas são estruturas quase tão simples quanto pilhas. Elas também são listas, e permitem inserções no final e remoções no começo. Mais precisamente, filas de acesso aleatório permitem as seguintes operações:

- Queue()
  Devolve uma fila vazia.
- Enqueue(q, x)
  Devolve uma cópia da fila q com o valor x inserido em seu fim.
- Dequeue(q)
  Devolve uma cópia da fila q com seu primeiro elemento removido.
- Size(q)
  Devolve o número de elementos na fila q.
- First(q)
  Devolve o primeiro elemento da fila q.

	Representação Binária	Skew-Binary
QUEUE()	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$
ENQUEUE(q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
DEQUEUE(q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
SIZE(q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
FIRST(q)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$
K-TH(q, k)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 3.2: Consumo de tempo e espaço da implementação de uma fila de acesso aleatório persistente, usando cada uma das implementações de pilha persistente, onde n é o tamanho atual da fila.

- K-TH(q, k)
  Devolve o k-ésimo elemento da fila q.

É possível, como com pilhas, implementar uma fila usando lista ligada. A lista torna-se uma árvore quando se adiciona a persistência. Seria então necessário usar a solução para o problema do Ancestral de Nível nas operações K-TH(q, k) e LAST(q).

Utilizaremos, entretanto, as próprias funções de pilha para implementar uma fila. Uma fila é representada por um par em que o primeiro elemento é uma pilha e o segundo é o número de elementos que já foram removidos da fila. No pseudocódigo, usaremos que (p, r) cria um par ordenado contendo a pilha p e um inteiro r, e todo par tem os campos *stack* e *rem* contendo cada um de seus elementos. Assumimos que criar pares, e devolvê-los por função, consome tempo constante.

Código 3.3: Fila de acesso aleatório persistente.

```
1:  function QUEUE()
2:      return (null, 0)
3:  function ENQUEUE(q, x)
4:      return (PUSH(q.stack, x), q.rem)
5:  function DEQUEUE(q)
6:      return (q.stack, q.rem + 1)
7:  function SIZE(q)
8:      return SIZE(q.stack) − q.rem                ▷ Função homônima da pilha
9:  function FIRST(q)
10:     return K-TH(q.stack, q.rem + 1)             ▷ Função homônima da pilha
11: function K-TH(q, k)
12:     return K-TH(q.stack, q.rem + k)             ▷ Função homônima da pilha
```

O Código 3.3 mostra a implementação das operações da fila. A correção segue diretamente da correção das funções para pilha. De fato, podendo acessar qualquer elemento de uma pilha, implementar uma fila é simples, já que usamos apenas as operações PUSH e K-TH, e simplesmente ignoramos os elementos que existem antes do começo da fila.

Desde que a implementação da pilha seja persistente, a implementação da fila é persistente pois nunca modifica nenhum par, apenas cria novos objetos. A Tabela 3.2 mostra o consumo de tempo e espaço da implementação discutida neste capítulo.

Capítulo 4

Deque com LA e LCA

Uma *fila duplamente terminada* ou fila de duas pontas é uma estrutura de dados que generaliza pilhas e filas. Essa estrutura é uma lista em que é possível adicionar e remover elementos de qualquer uma das suas extremidades. Ela é usualmente chamada de deque, uma abreviatura de dequeue.

Uma deque de acesso aleatório é uma lista que permite as seguintes operações:

- Deque()
  Devolve uma deque vazia.
- PushFront(d, x)
  Devolve uma cópia da deque d com o valor x inserido no seu início.
- PushBack(d, x)
  Devolve uma cópia da deque d com o valor x inserido no seu fim.
- Front(d)
  Devolve o primeiro elemento de d.
- Back(d)
  Devolve o último elemento de d.
- PopFront(d)
  Devolve uma cópia da deque d sem o primeiro elemento.
- PopBack(d)
  Devolve uma cópia da deque d sem o último elemento.
- k-th(d, k)
  Devolve o k-ésimo elemento da deque d.

Note que, usando apenas PushBack(d, x), Back(d) e PopBack(d), simulamos uma pilha. Analogamente, usando apenas PushBack(d, x), Front(d) e PopFront(d), simulamos uma fila simples.

Apresentaremos uma implementação persistente de deques, que se baseia nas implementações de pilhas e filas persistentes discutidas no Capítulo 3. Esta implementação utiliza Ancestral de Nível e Ancestral Comum Mais Profundo, discutidos nos Capítulos 1 e 2.

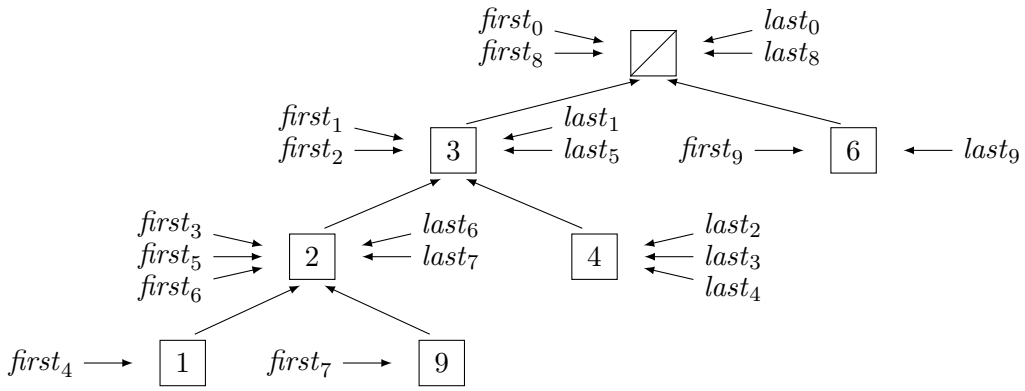

Figura 4.1: Arborescência criada pela sequência de operações do Exemplo 4.1.

$(first_0, last_0) = \text{DEQUE}()$	$\Rightarrow$
$(first_1, last_1) = \text{PUSHBACK}((first_0, last_0), 3)$	$\Rightarrow 3$
$(first_2, last_2) = \text{PUSHBACK}((first_1, last_1), 4)$	$\Rightarrow 3\ 4$
$(first_3, last_3) = \text{PUSHFRONT}((first_2, last_2), 2)$	$\Rightarrow 2\ 3\ 4$
$(first_4, last_4) = \text{PUSHFRONT}((first_3, last_3), 1)$	$\Rightarrow 1\ 2\ 3\ 4$
$(first_5, last_5) = \text{POPBACK}((first_3, last_3))$	$\Rightarrow 2\ 3$
$(first_6, last_6) = \text{POPBACK}((first_5, last_5))$	$\Rightarrow 2$
$(first_7, last_7) = \text{PUSHFRONT}((first_6, last_6), 9)$	$\Rightarrow 9\ 2$
$(first_8, last_8) = \text{POPFRONT}((first_6, last_6))$	$\Rightarrow$
$(first_9, last_9) = \text{PUSHFRONT}((first_8, last_8), 6)$	$\Rightarrow 6$

Exemplo 4.1: Exemplo de uso de uma deque persistente.

Esta implementação, e a da fila no Capítulo 3, são de minha autoria. Ambas são baseadas na implementação da pilha de acesso aleatório persistente. Desenvolvi inicialmente a ideia para uma fila persistente ao escrever um problema para a Seletiva USP de 2016, uma prova que seleciona alunos da USP para participar da Maratona de Programação. O problema em inglês pode ser acessado em `codeforces.com/gym/101064/problem/G`.

Acreditamos que, mesmo sendo necessário resolver o problema do Ancestral de Nível e Ancestral Comum Mais Profundo, esta implementação é mais simples que as outras, apresentadas nos capítulos seguintes.

4.1 Representação e visão geral

Na implementação de pilhas no Capítulo 3, a lista ligada, quando em um contexto persistente, se torna uma arborescência. É possível adicionar novas folhas, pois estas não mudam os ponteiros dos outros elementos; e então é possível adicionar elementos ao final da pilha.

Na implementação de filas, no mesmo capítulo, também foi discutido como simular a remoção de elementos do início da estrutura, guardando, junto com o nó, o número de elementos já removidos. Dessa forma, os elementos da fila são um sufixo do caminho da raiz até o vértice. Isso, porém, não permite adicionar elementos no início.

Para a implementação de deques, ainda usaremos uma arborescência, ou seja, as arestas são em direção à raiz, mas cada versão de uma deque será associada a dois nós, *first* e *last*. Os elementos

da deque serão os elementos no caminho de *first* para *last*, se considerarmos que as arestas não têm direção. Na Figura 4.1, a deque apontada pelo par $(first_3, last_3)$, por exemplo, contém os elementos 2, 3 e 4.

Para adicionar um elemento no início da deque, criamos uma nova folha conectada a *first* e substituímos *first* por esta folha; para adicionar um elemento no final da deque, criamos uma nova folha conectada a *last* e substituímos *last*. Criar folhas em uma árvore não muda os caminhos entres vértices já existentes, então as versões de deques anteriores continuam válidas, e o novo par $(first, last)$ representa a deque desejada, com um elemento adicionado em seu início ou final. Na Figura 4.1 ao inserir 1 no começo da deque representada por $(first_3, last_3)$, obtemos a deque $(first_4, last_4)$.

Para remover um elemento do começo, alteramos apenas *first* para apontar para o segundo elemento no caminho de *first* para *last*; para remover um elemento do final, alteramos *last* para apontara para o penúltimo elemento do caminho de *first* para *last*. Como fazer essas operações será discutido nas próximas seções. Na Figura 4.1 ao remover o último elemento da deque $(first_5, last_5)$ obtemos a deque $(first_6, last_6)$.

Devemos ter uma representação especial para deques vazias, por exemplo o valor **null** em *first* e *last*. Ao remover um elemento de uma deque com um único elemento (quando $first = last$), devolvemos este nó vazio; e para adicionar algum elemento à deque vazia basta criamos e devolvemos um novo nó que aponta para **null**. Dessa forma nossa estrutura é na verdade uma coleção de arborescências, ou podemos considerar que é uma arborescência em que a raiz é **null**, assim como fizemos para pilhas no Capítulo 3. Na Figura 4.1, ao remover o único elemento da deque $(first_6, last_6)$, obtemos a deque vazia $(first_8, last_8)$, e ao adicionar 6 a essa deque obtemos $(first_9, last_9)$.

4.2 Acesso e inserção

Em cada nó dá arborescência guardaremos: o pai deste nó (o nó para o qual ele aponta), no campo *parent*; sua profundidade, no campo *depth*; e seu valor associado, que é o valor armazenado na deque, no campo *val*. Usamos que **new** NODE(x, p, d) cria um novo nó com valores x, p e d nos campos *val*, *parent* e *depth*, respectivamente. Esta função também realiza o pré-processamento necessário para o algoritmo de Ancestral de Nível funcionar, ou seja, chama a função ADDLEAF(u) para o novo vértice, como discutido nos Capítulos 1 e 2.

Uma deque é representada por um par ordenado de nós. Usaremos (a, b) para criar uma deque associada aos nós a e b, e estes nós podem ser acessados pelos campos *first* e *last*.

O Código 4.1 mostra a implementação das operações mais simples. A função SWAP(d) devolve o inverso da deque d (basta trocar seus dois vértices), e é usada para diminuir a necessidade de duplicar código entre as operações PUSHFRONT e PUSHBACK, assim como POPFRONT e POPBACK.

As implementações dadas são funcionais e mantêm a propriedade que os elementos de uma deque são os valores dos nós no caminho de seu campo *first* para *last*.

Código 4.1: Operações de acesso e inserção

```
1: function Deque()
2:     return (null, null)
3: function Swap(d)
4:     return (d.last, d.first)
5: function PushFront(d, x)
6:     if d.first = null :
7:         u = new Node(x, null, 1)
8:         return (u, u)
9:     else
10:        return (new Node(x, d.first, d.first.depth + 1), d.last)
11: function PushBack(d, x)
12:    return Swap(PushFront(Swap(d), x))
13: function Front(d)
14:    return d.first.val
15: function Back(d)
16:    return d.last.val
```

4.3 Remoção

Discutiremos como determinar o segundo elemento no caminho de *first* para *last*, já que encontrar o penúltimo é encontrar o segundo no caminho de *last* para *first*, e usaremos Swap para não ter que tratar esse caso separadamente. Para uma árvore com raiz, dizemos que o *ancestral comum mais profundo* de *first* e *last* é o ancestral comum de ambos com maior profundidade. Seja *mid* este ancestral. Então o caminho de *first* para *last* é o caminho de *first* para *mid* concatenado com o caminho de *mid* para *last*.

O caminho de *first* para *mid* é $(first, first.parent, \ldots, mid)$; então se $first \neq mid$ temos que $first.parent$ é o segundo elemento do caminho. Se $first = mid$ então o caminho de *first* para *last* é $(first, \ldots, last.parent, last)$. Este caminho tem tamanho $last.depth - first.depth$, e então o segundo nó deste caminho é o $(last.depth - first.depth - 1)$-ésimo ancestral de *last*, considerando que o 0-ésimo ancestral de um vértice é ele mesmo, primeiro é seu pai, e assim por diante.

Código 4.2: Operações de remoção

```
1: function PopFront(d)
2:     if d.first = d.last :
3:         return Deque()
4:     else if LCA(d.first, d.last) = d.first :
5:         return (LevelAncestor(d.last.depth − d.first.depth − 1, d.last), d.last)
6:     else
7:         return (d.first.parent, d.last)
8: function PopBack(d)
9:     return Swap(PopFront(Swap(d)))
```

O Código 4.2 mostra a implementação das operações PopFront e PopBack. Usamos o predicado $\text{LCA}(d.first, d.last) = d.first$ para determinar se $d.first$ é o ancestral comum mais profundo de $d.first$ e $d.last$, mas esta condição pode ser verificada usando apenas chamadas

	Representação binária	Skew-binary
Deque()	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$
PushFront(q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
PushBack(q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
Front(q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Back(q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
PopFront(q)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(1)$
PopBack(q)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(1)$
k-th(q, k)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 4.1: Consumo de tempo e espaço da implementação de uma deque de acesso aleatório persistente, usando cada uma das implementações de Ancestral de Nível, onde n é o tamanho da deque.

para LevelAncestor. Como discutido anteriormente, se isto ocorre, então o caminho de *first* para *last* é $(first, \ldots, last.parent, last)$, mas neste caso $d.first$ é o $(last.depth - first.depth)$-ésimo ancestral de *last*. Então

$$\text{LCA}(d.first, d.last) = d.first$$

é equivalente a

$$d.first.depth < d.last.depth \textbf{ and } \text{LevelAncestor}(d.last.depth - d.first.depth, d.last) = d.first.$$

4.4 Acesso a outros elementos

Para implementar a operação k-th(d, k), é necessário comparar $k-1$ com o tamanho do caminho de *first* até *mid*, o ancestral comum mais profundo de *first* e *last*. Esse caminho tem tamanho $\ell_1 := first.depth - mid.depth$. Se $k-1$ for menor ou igual a ℓ_1, o k-ésimo elemento da deque é o $(k-1)$-ésimo ancestral de *first*; senão, devemos determinar o $(k-\ell_1)$-ésimo elemento da segunda parte do caminho. Seja $\ell_2 := last.depth - mid.depth$ o tamanho do caminho de *mid* até *last*. Então o $(k-\ell_1)$-ésimo elemento da segunda parte do caminho é o $(\ell_2-(k-1-\ell_1)) = (\ell_2+\ell_1+1-k)$-ésimo ancestral de *last*.

Código 4.3: Operação k-th

```
1: function k-th(d, k)
2:     mid = LCA(d.first, d.last)
3:     ℓ1 = d.first.depth − d.mid.depth
4:     ℓ2 = d.last.depth − d.mid.depth
5:     if k − 1 ≤ ℓ1 :
6:         return LevelAncestor(k − 1, d.first)
7:     else
8:         return LevelAncestor(ℓ1 + ℓ2 + 1 − k, d.last)
```

O Código 4.3 faz exatamente o que discutido, e portanto corretamente devolve o k-ésimo elemento da deque, se k é válido, ou seja, está entre 1 e o tamanho da deque. A Tabela 4.1 mostra o consumo de tempo e espaço da implementação discutida nesse capítulo, usando o Ancestral de Nível do Capítulo 1 e 2.

Capítulo 5

Deque recursiva

Neste capítulo, apresentaremos um implementação persistente desta estrutura. A implementação é discutida por Kaplan [7] e serve como base para a implementação discutida no Capítulo 6.

5.1 Representação

A implementação usual de uma deque usando lista ligada utiliza uma lista duplamente ligada, e então para adicionar um elemento é necessário *mudar* campos de nós já existentes da lista, o que não resulta numa implementação persistente como ocorreu com a pilha. Uma solução é fazer como na implementação do Capítulo 4, que transforma esta lista em uma arborescência. Neste capítulo, trataremos isto de outra forma.

Para conseguirmos manter a implementação funcional, mudaremos como uma deque é representada, com um esquema de representação recursiva. Para facilitar a descrição, é necessário deixar claro o conjunto de elementos que a deque pode armazenar. Seja T esse conjunto, ou seja, operações de PUSH recebem elementos de T como argumento e operações de POP devolvem elementos desse conjunto.

Uma deque persistente que armazena elementos do conjunto T é um nó com três campos opcionais: *prefix*, *center*, e *suffix*; além de um campo obrigatório *size*. Um campo ser opcional significa que ele pode não estar presente. Quando não estiver presente, o campo assume o valor **null**. Os campos *prefix* e *suffix* armazenam valores de T, o campo *center* armazena uma deque persistente que armazena elementos do conjunto $T \times T$, ou seja, pares ordenados de T, e o campo *size* armazena o tamanho da deque.

Como indicado pelos nomes, *prefix* armazena um prefixo da sequência de elementos que é a deque, *suffix* representa um sufixo, e *center* os elementos do meio. A Figura 5.1 mostra uma possível deque persistente, cuja sequência é $(2, 3, 4, 5, 6, 7, 8)$. Note que os campos **null** são ignorados.

Observe que, para uma deque não vazia, considerando que nenhum campo *center* guarda uma deque vazia, o número máximo de níveis nessa estrutura é $\lg n$, onde n é o número de elementos na deque, já que no i-ésimo nível cada elemento *prefix* e *suffix* armazena 2^{i-1} elementos.

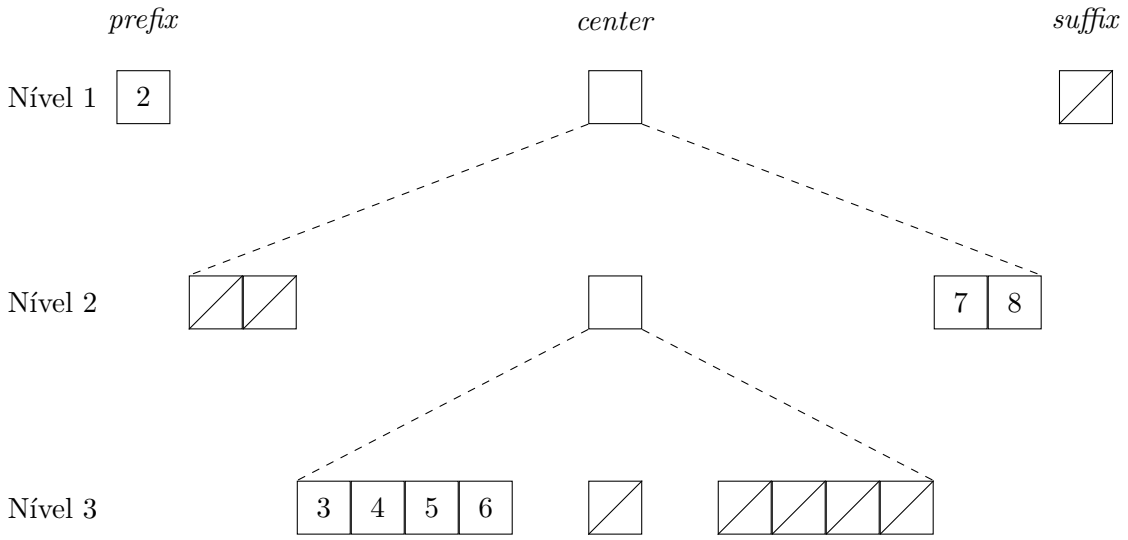

Figura 5.1: Deque persistente

Código 5.1: Operações de acesso para uma deque persistente.

```
1: function Deque()
2:     return null
3: function Front(d)
4:     if d.prefix ≠ null :
5:         return d.prefix
6:     else if d.center = null :
7:         return d.suffix
8:     else
9:         return Front(d.center).first
10: function Back(d)
11:     if d.suffix ≠ null :
12:         return d.suffix
13:     else if d.center = null :
14:         return d.prefix
15:     else
16:         return Back(d.center).second
```

5.2 Operações de acesso

Devido à forma recursiva como a estrutura foi definida, recursão é muito útil para lidar com esta estrutura. Assumimos sempre que todas as operações chamadas são válidas, ou seja, FRONT(d) e BACK(d) são chamadas apenas quando d não é vazia, e consequentemente seu valor não é **null**. Veja o Código 5.1. Para questões de implementação, os elementos de $T \times T$ são representados como um par com campos *first* e *second*, ou seja, se $a, b \in T$ então $(a, b) \in T \times T$ e vale que $(a, b).first = a$ e $(a, b).second = b$.

Vamos analisar a operação FRONT(d). Pela maneira como definimos a deque, se *prefix* não é nulo então ele é um prefixo de d, logo este é a resposta de FRONT(d). Se não, se *center* não é nulo, como não guardamos deques vazias, este tem algum elemento. O valor devolvido é então o primeiro elemento de $d.center$, e como esta é uma deque de pares de elementos do tipo $T \times T$, temos que o primeiro elemento do primeiro par de $d.center$ é o primeiro elemento de d. Caso contrário, o único elemento da deque é $d.suffix$ e o valor devolvido também é correto. Portanto, a função FRONT(d) funciona corretamente.

Note que BACK(d) é simétrica a FRONT(d). Basta trocar *prefix* com *suffix* e *first* com *second*, o mesmo ocorrerá com as outras operações, pela simetria da estrutura. Então nas próximas seções apenas a versão FRONT de PUSH e POP será detalhada.

Ambas as funções apresentadas consomem tempo $\mathcal{O}(\lg n)$, onde n é o tamanho da deque, e não modificam a estrutura.

5.3 Operações de modificação

Ao adicionar um elemento x no início da deque d, se $d.prefix$ for nulo, podemos colocar x no campo *prefix*, e então a deque continua válida e com x no início. Se $d.prefix$ não for nulo, podemos juntá-lo com x e então inserir o par $(x, d.prefix)$ em $d.center$, e tornar $d.prefix$ nulo, assim a deque continua válida e com x no início.

Analogamente, ao remover um elemento do início de d, se $d.prefix$ não for nulo, podemos remover esse elemento, caso contrário, podemos remover um elemento de $d.center$ e colocar apenas o segundo elemento em $d.prefix$, já que os elementos de $d.center$ são pares, removendo assim o primeiro elemento da deque. Note que precisamos do elemento removido, então usaremos uma função auxiliar POPFRONTAUX(d) que devolve um par com o primeiro elemento de d e uma cópia de d sem este elemento.

Se fizermos as operações desta forma, estaremos mudando a estrutura, que não será funcional. Note que apenas mudamos um prefixo dos níveis, e cada nível consiste de apenas um número constante de dados (se considerarmos que os pares são na verdade ponteiros para pares). Assim, se apenas copiarmos todos os nós que iremos modificar, mantemos a estrutura funcional. Assumimos que **new** NODE(pr, ct, sf, sz) devolve um nó como discutido, com *prefix*, *center*, *suffix* e *size* inicializados com pr, ct, sf e sz, respectivamente. Veja o Código 5.2.

Proposição 5.1. *A operação* PUSHFRONT(d, x) *funciona corretamente, devolvendo uma cópia de* d *com* x *inserido em seu início, sem modificar* d.

Demonstração. A prova será feita por indução no número de níveis da estrutura recursiva da deque.

Código 5.2: Operações de modificação para uma deque.

```
1: function PushFront(d, x)
2:     if d = null :
3:         return new Node(x, null, null, 1)
4:     else if d.prefix = null :
5:         return new Node(x, d.center, d.suffix, d.size + 1)
6:     else
7:         return new Node(null, PushFront(d.center, (x, d.prefix)), d.suffix, d.size + 1)
8: function PopFront(d)
9:     return PopFrontAux(d).second
10: function PopFrontAux(d)
11:     if d.prefix ≠ null and d.center = null and d.suffix = null :
12:         return (d.prefix, Deque())
13:     else if d.prefix ≠ null :
14:         return (d.prefix, new Node(null, d.center, d.suffix, d.size − 1))
15:     else if d.center = null :
16:         return (d.suffix, Deque())
17:     else
18:         (x, c) = PopFrontAux(d.center)
19:         return (x.first, new Node(x.second, c, d.suffix, d.size − 1))
```

A base se dá quando d é nulo, nesse caso a deque é vazia e a linha 3 devolve uma deque com x em seu campo *prefix* e os outros campos nulos. Assuma então que a proposição vale para todas as deques com número de níveis menor que d.

Se $d.prefix$ é nulo, podemos armazenar x nessa posição e teremos adicionado x como primeiro elemento. Como não podemos modificar os campos de d, criamos um novo nó com campo *prefix* como x e os outros campos iguais aos de d. Esse caso é então tratado corretamente nas no **if** das linhas 4 e 5.

Caso contrário, juntamos x e $d.prefix$ em um par e o adicionamos recursivamente no começo de $d.center$. Como a deque $d.center$ tem menos níveis que d, a hipótese de indução vale e a operação funciona nesse caso. □

De maneira simétrica temos que a operação PushBack(d, x) também está correta. Provaremos a seguir que PopFrontAux(d) funciona corretamente, e portanto PopFront(d) também, já que apenas devolve a deque devolvida por PopFrontAux(d). Consequentemente, PopBack(d) está correta.

Proposição 5.2. *A função* PopFrontAux(d) *funciona corretamente, devolvendo, sem modificar d, o primeiro elemento de d e uma cópia de d sem este elemento.*

Demonstração. A prova também será feita por indução no número de níveis da estrutura da deque.

Se $d.prefix$ não é nulo, basta remover e devolver este elemento, assim como uma cópia da deque sem ele. Se $d.prefix$ era o único nó da estrutura, devolve-se uma deque vazia. Este caso é tratado na linha 11. Caso contrário, o **if** das linhas 13 e 14 devolve $d.prefix$ e cria um novo nó que não tem esse campo, sem modificar d, portanto trata corretamente este caso.

Caso contrário, se $d.center$ é nulo, sabemos que d é uma deque com apenas um elemento, armazenado em $d.suffix$, pois sabemos que PopFrontAux(d) não é chamada se d é vazia. A

linha 16 trata corretamente esse caso, devolvendo $d.suffix$ e uma nova deque vazia. A base da indução, quando a deque tem um nível, sempre satisfaz um destes casos, logo é tratada corretamente.

No último caso, usamos PopFrontAux($d.center$), que está correta pois $d.center$ tem menos níveis que d, para remover recursivamente o primeiro elemento de $d.center$. Esse elemento é um par de elementos de d. Retornamos o primeiro elemento do par, que é o primeiro elemento de d, e colocamos o segundo elemento no campo *prefix*, removendo assim o primeiro elemento da deque. Também é necessário substituir *center* pela nova deque devolvida pela chamada recursiva. A linha 19 devolve então um novo nó com os campos corretos. □

Ambas operações consomem tempo e espaço $\mathcal{O}(\lg n)$, onde n é o tamanho da deque, pois apenas percorrem a estrutura dos nós, que tem altura $\mathcal{O}(\lg n)$, e realizam operações que consomem tempo e espaço constante por nível.

5.4 Acesso a outros elementos

Para tornar a deque de acesso aleatório, resta implementar K-TH(d, k). Veja o Código 5.3.

Código 5.3: Implementação de K-TH(d, k).

```
1:  function K-TH(d, k)
2:      if k = 1 and d.prefix ≠ null :
3:          return d.prefix
4:      if k = d.size and d.suffix ≠ null :
5:          return d.suffix
6:      if d.prefix ≠ null :
7:          k = k − 1
8:      if k is odd :
9:          return K-TH(d.center, ⌈k/2⌉).first
10:     else
11:         return K-TH(d.center, ⌈k/2⌉).second
```

Proposição 5.3. *A função* K-TH(d, k) *devolve o k-ésimo elemento de d, para $k \leq d.size$.*

Demonstração. As linhas 3 e 5 tratam o caso em que estamos buscando, respectivamente, o primeiro e o último elemento, e esses estão no prefixo ou sufixo. Caso contrário, o k-ésimo elemento está em $d.center$ e a linha 7 arruma k para indicar qual elemento de $d.center$ deve ser buscado, já que, se $d.prefix$ é não nulo, temos que encontrar o $(k-1)$-ésimo elemento dos que estão em $d.center$.

Como $d.center$ é uma deque de pares, usamos K-TH($d.center$, $\lceil \frac{k}{2} \rceil$) para encontrar o $\lceil \frac{k}{2} \rceil$-ésimo par desta deque, pois o primeiro par guarda os elementos 1 e 2, o segundo guarda os elementos 3 e 4, e assim por diante. Se k é ímpar, queremos o primeiro elemento desse par, senão o segundo. De fato, se k é ímpar, o par tem os elementos com índice k e $k+1$, se não tem os elementos com índice $k-1$ e k. □

A operação consome tempo $\mathcal{O}(\lg n)$, onde n é o tamanho da deque, pois este é o número máximo de níveis da estrutura recursiva. A Tabela 5.1 mostra o consumo de tempo e espaço da implementação discutida neste capítulo.

	Tempo/Espaço
Deque()	$\mathcal{O}(1)/\mathcal{O}(1)$
PushFront(q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
PushBack(q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
Front(q)	$\mathcal{O}(\lg n)$
Back(q)	$\mathcal{O}(\lg n)$
PopFront(q)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
PopBack(q)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
k-th(q, k)	$\mathcal{O}(\lg n)$

Tabela 5.1: Consumo de tempo e espaço da implementação deste capítulo, onde n é o tamanho da deque.

Capítulo 6

Deque de Kaplan e Tarjan

Modificaremos a implementação do Capítulo 5 para que todas as operações, exceto K-TH, consumam tempo e espaço $\mathcal{O}(1)$. Esta implementação foi inicialmente apresentada por Kaplan e Tarjan [8].

6.1 Contadores binários

Deque persistente como um contador binário

Note que, na estrutura do Capítulo 5, se adicionarmos elementos apenas usando PUSHFRONT(d, x), o primeiro nível da estrutura é visitado em todas as adições, o segundo nível é visitado a cada duas adições, o terceiro a cada quatro, e assim por diante.

Esse comportamento é similar ao de um contador binário. Podemos considerar que cada nível é um dígito: 0 se *prefix* é nulo ou 1 se *prefix* é não nulo. O primeiro nível é o dígito menos significativo. Assim, adicionar um elemento usando PUSHFRONT(d, x) é equivalente a somar 1 no contador binário, pois se encontramos um 0 colocamos x ali, senão, formamos um par de x com o elemento que ali estava (transformando essa posição em 0), e adicionamos esse par na posição seguinte (somamos 1 na posição seguinte).

Apesar de sabermos que adições em um contador binário consomem tempo amortizado $\mathcal{O}(1)$, sabemos que algumas adições específicas podem levar tempo $\Theta(\lg n)$. Então, como estamos em uma estrutura persistente, podemos efetuar essa operação repetidas vezes na mesma versão, e não é possível manter o custo amortizado $\mathcal{O}(1)$.

Contadores binários redundantes

Contadores binários redundantes são contadores binários nos quais cada dígito pode ser 0, 1 ou 2. Ainda como em um contador normal, se o contador redundante tem n dígitos $x_0, x_1, \ldots, x_{n-1}$, então o seu valor é $\sum_{i=0}^{n-1} x_i 2^i$. Ao permitirmos dígitos 2, passam a existir diversas formas de representar o mesmo número. Por exemplo, o número 9 pode ser representado como 1001, 201 ou 121.

Dizemos que um contador binário redundante x é *semirregular* se, entre quaisquer dígitos 2, existe um dígito 0, ou seja, se $x_i = 2$, $x_j = 2$ e $i < j$, então existe k tal que $x_k = 0$ e $i < k < j$. Se também existe um 0 entre o primeiro 2 e o começo do contador (dígito menos significativo), dizemos

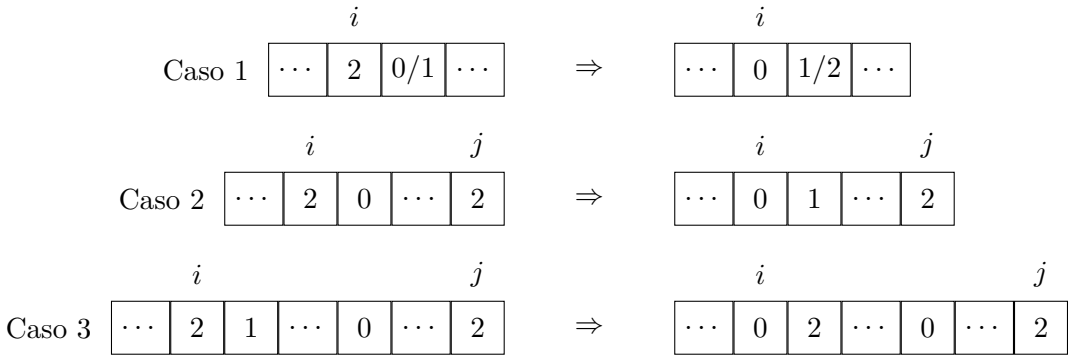

Figura 6.1: Casos possíveis para o procedimento FIX.

que o contador é *regular*. Um contador com apenas 0s é claramente regular, e vamos mostrar como somar 1 em um contador binário regular mudando apenas $\mathcal{O}(1)$ dígitos e mantendo sua regularidade.

Suponha que temos um contador binário regular, e queremos somar 1. Pela regularidade, $x_0 < 2$, e portanto é possível aumentar esse dígito sem causar um estouro. Ao fazer isso, o contador pode deixar de ser regular, apesar de ainda ser semirregular, pois é possível que não exista mais nenhum 0 antes do primeiro dígito 2. Encontramos então o primeiro dígito 2, se existir, e realizamos um procedimento FIX nele. Se não existir nenhum 2, o contador é regular.

Seja i o índice do primeiro 2. Um procedimento FIX consiste em transformar esse dígito em 0 e adicionar 1 no dígito $i + 1$. Se isso for possível, é claro que o valor representado pelo contador continua o mesmo, pois $2^i x_i + 2^{i+1} x_{i+1} = 2^i(x_i - 2) + 2^{i+1}(x_{i+1} + 1)$.

A Figura 6.1 mostra os casos possíveis em que se realiza o procedimento FIX. Nesta figura, "..." representa zero ou mais dígitos 0 ou 1. A esquerda representa como eram os dígitos antes e a direita, depois do procedimento. O dígito na posição j, nos Casos 2 e 3, é o segundo dígito 2. À direita do último dígito representado, temos que o contador é semirregular. Como no máximo removemos o primeiro 0 do contador, é garantido que existe um 0 entre o primeiro e, se existir, o segundo 2.

No Caso 1, o resultado ou não tem dígito 2, ou tem um dígito 2 com um 0 imediatamente à sua esquerda, logo o contador continua regular. No Caso 2, quando o dígito $i + 1$ é 0, o resultado tem o dígito i como 0, e o dígito j como o primeiro 2, logo a regularidade é respeitada. No Caso 3, o dígito $i + 1$ passa a ser o primeiro 2, mas como ele era anteriormente 1, ainda existe um 0 antes da posição j. Além disso, o dígito i é 0, logo a regularidade é respeitada.

O contador portanto continua regular, mudando no máximo 3 dígitos. Para encontrar o primeiro 2 em tempo constante, podemos manter uma pilha com todos os índices de dígitos que são 2. Vamos esboçar um código para fazer essa adição. Supomos que temos uma pilha p implementada como a do Capítulo 3, e que o contador é armazenado em um vetor x, indexado a partir de 0.

A função ADD1(x, p) no Código 6.1 adiciona 1 ao contador redundante regular x que tem os índices de seus dígitos 2 armazenados em p.

6.2 Visão geral

Como discutido na Seção 6.1, a implementação de deque apresentada no Capítulo 5 tem uma certa relação com contadores binários. Iremos então usar a ideia de contadores binários redundantes,

Código 6.1: Adicionando 1 em contador binário regular.

```
1:  function ADD1(x, p)
2:      x_0 = x_0 + 1
3:      if x_0 = 2 :
4:          p = PUSH(p, 0)
5:      if p ≠ null :                          ▷ FIX no primeiro dígito 2
6:          i = TOP(p)
7:          p = POP(p)
8:          x_i = 0
9:          x_{i+1} = x_{i+1} + 1
10:         if x_{i+1} = 2 :
11:             p = PUSH(p, i + 1)
12:     return (x, p)
```

também discutidos na Seção 6.1, para modificar a implementação da deque persistente para que qualquer operação da deque, exceto K-TH, consuma tempo $\mathcal{O}(1)$.

A representação se mantém parecida. Cada nível contém um prefixo e um sufixo da deque, com o nível $i+1$ armazenando pares de elementos do nível i. Os prefixos e sufixos, porém, são deques (não persistentes) que podem conter até 5 elementos.

Cada nível será associado a um dígito 0, 1 ou 2, assim como um contador binário redundante. Durante o algoritmo, o contador formado pelos dígitos associados a cada nível é regular, considerando que o primeiro nível é o dígito menos significativo. Para garantir isso, usamos uma adaptação do procedimento FIX discutida anteriormente. Note que o valor do contador não tem relação nenhuma com a estrutura, apenas usamos sua regularidade para conseguir aplicar todas as operações em tempo constante. Como veremos, a adaptação do procedimento FIX pode até mudar o valor do contador, diferente da discutida anteriormente.

Para cada nível, associamos suas sub-deques (prefixo e sufixo) a algum dígito. O dígito associado ao nível é então o máximo entre estes dois dígitos. O dígito de uma sub-deque é 0 se essa tem 2 ou 3 elementos, 1 se tem 1 ou 4 elementos, e 2 se tem 0 ou 5 elementos. Intuitivamente, esses dígitos estão associados a quanto "espaço livre" uma sub-deque tem, tanto para adicionar quanto para remover elementos. Note que, se o dígito é 2, a sub-deque tem 0 ou 5 elementos, então pode não ser possível remover ou adicionar algum elemento. De forma similar, se o dígito é 0, é sempre possível adicionar ou remover até 2 elementos desta sub-deque. Como escolhemos o máximo entre os dois dígitos, temos que, se um nível tem dígito pequeno, é possível adicionar ou remover elementos de seu prefixo e sufixo.

Assim como na primeira implementação, pode ser necessário tratar separadamente o último nível, se este tem apenas seu prefixo ou sufixo. Nesse caso, o dígito associado a esse nível é o dígito de sua sub-deque não vazia. Isso se deve ao fato que, nesse caso, a sub-deque não vazia é tanto seu prefixo quanto seu sufixo. Dizemos que esse caso é degenerado.

6.3 Regularidade e operações

Uma deque é *regular* ou *semirregular* quando os dígitos de seus níveis formam um contador regular ou semirregular, respectivamente. Vamos garantir que a deque devolvida por qualquer uma

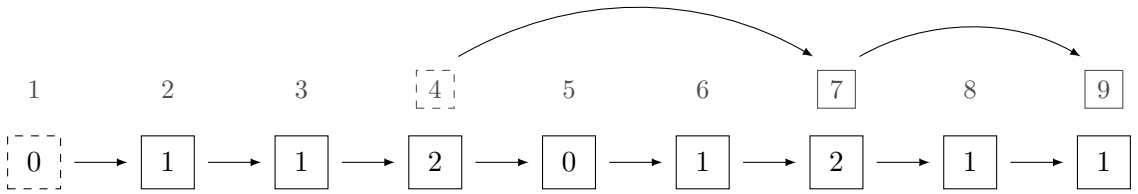

Figura 6.2

das operações é regular.

Assuma que a deque é regular. Para realizar uma operação de Push ou Pop, note que, devido à regularidade, o primeiro nível tem dígito 0 ou 1, logo ambas suas sub-deques tem dígito 0 ou 1, e então é possível adicionar ou remover um elemento de qualquer uma dessas sub-deques. Se estamos no caso degenerado, também é possível adicionar ou remover um elemento de sua sub-deque não vazia, o que é suficiente nesse caso. Após adicionar ou remover um elemento, o dígito associado ao primeiro nível pode ser 0, 1 ou 2 (ele pode diminuir, se manter, ou aumentar). É possível que a deque não seja mais regular, passando a ser apenas semirregular; nesse caso, realizamos um procedimento Fix no primeiro nível que tem dígito 2. Uma diferença dos contadores binários analisados anteriormente é que realizamos um Fix apenas quando a estrutura não é mais regular.

Um procedimento Fix no nível i com dígito 2 transforma esse dígito em 0 e aumenta em no máximo 1 o dígito do próximo nível. Pela semirregularidade da estrutura, o nível $i+1$ tem dígito 0 ou 1. Para cada uma das sub-deques do nível i, se esta sub-deque tem pelo menos quatro elementos, removemos dois destes, formamos um par com estes e o inserimos na correspondente sub-deque do nível $i+1$; se a sub-deque tem no máximo um elemento, removemos um par da sub-deque do nível $i+1$, e inserimos os dois elementos do par na sub-deque. Dessa forma, ao final do Fix, cada uma das sub-deques do nível i tem 2 ou 3 elementos, então o dígito do nível i é 0.

Quando dizemos "remover" um elemento de uma sub-deque e "inserir" em outra, isto deve ser feito na extremidade correta. Por exemplo, remove-se elementos do *final* do prefixo de i para adicionar um par no *início* do prefixo de $i+1$. O procedimento Fix foi descrita de forma simples, mas na prática é necessário tratar muitos casos, que serão descritos a seguir. É importante notar que Fix é o mesmo, não importando se o procedimento que o chamou foi Push ou Pop. Após um Fix em uma estrutura semirregular, esta se torna regular, pelos mesmos argumentos discutidos na Seção 6.1.

6.4 Pilha de pilhas, e implementações funcionais

A primeira dificuldade é encontrar (e modificar) o primeiro nível com dígito 2, pois este nível pode estar arbitrariamente fundo na lista de níveis (ou seja, em um nível $\omega(1)$). Para a implementação dos contadores binários redundantes, usamos uma pilha com os índices dos dígitos que eram 2. Aquela implementação, porém, não era persistente, e temos que nos preocupar com isso nesta implementação. Assim como nas estruturas persistentes anteriores, vamos fazer uma implementação funcional.

Uma ideia inicial é manter uma pilha (persistente) de nós com dígitos 2, e, como nas implementações anteriores, sempre que for necessário modificar um campo de um nó, copiar esse nó.

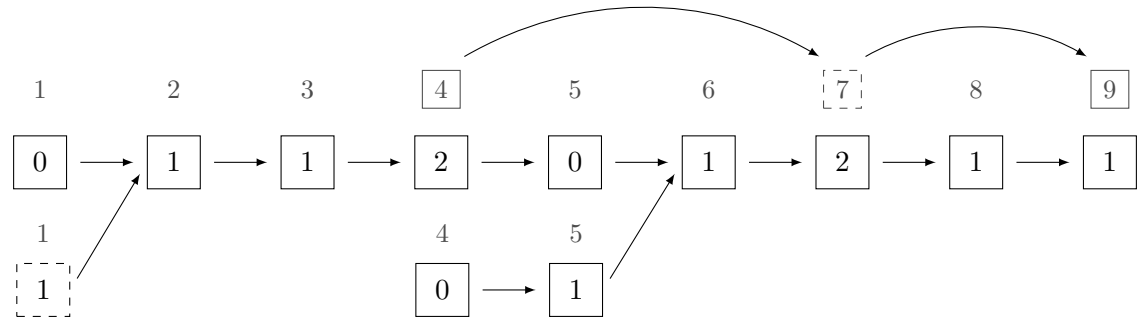

Figura 6.3

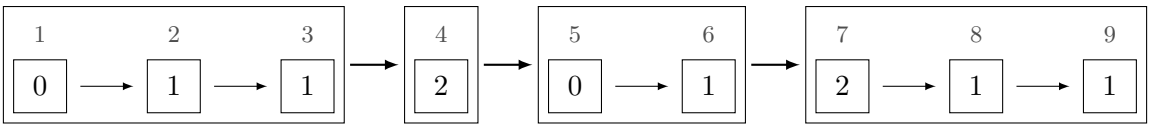

Figura 6.4

Isso, porém, não funciona.

Considere a Figura 6.2, onde os números indicam os dígitos correspondentes aos níveis, e os nós tracejados são os nós de entrada, ou seja, o primeiro nível e o topo da pilha de nós com dígito 2. Suponha que uma operação de PUSH ou POP modifique o dígito do primeiro nível para 1, e então seja realizado um procedimento FIX no primeiro dígito 2, no nível 4, e essa operação transforme o dígito em 0 e o dígito do nível 5 em 1.

Se copiarmos os nós modificados, a estrutura fica como na Figura 6.3, que é inválida. Ao acessar o nível 4 a partir do nível 1 (em, por exemplo, uma operação K-TH), acessamos na verdade a versão antiga do nível 4 (que tinha dígito 2), em vez da nova versão copiada. Isso acontece pois copiamos um nó que era apontado por algum nó não copiado. No caso, o nó de nível 4 era apontado pelo nó de nível 3, que não foi copiado nem modificado, e portanto ainda aponta para a versão antiga do nível 4.

Para implementar uma estrutura de forma funcional (e portanto persistente), é necessário copiar todos os nós que precisam ser modificados, e além disso garantir que *nenhum nó copiado é apontado por um nó não copiado*. Na nossa primeira implementação de deques persistentes, isso é sempre verdade pois copiamos um prefixo dos níveis, e cada nível apenas aponta para o próximo nível.

Para consertar a implementação, precisamos garantir que os nós copiados não são apontados por nós não copiados. Para isso, separa-se os níveis em várias pilhas; em cada pilha o nível do topo tem dígito diferente de 1, ou é o nível 1, e todos os outros elementos da pilha tem dígito 1. Faz-se então uma pilha em que cada elemento é o topo de uma destas sub-pilhas. Então, o procedimento FIX é sempre realizada no nível 1 ou no primeiro elemento da segunda pilha.

A Figura 6.4 mostra como seriam essas pilhas na mesma deque da Figura 6.2. Faremos a implementação destas pilhas adicionando um campo a mais a cada nó, *next*, que, para nós que são topos de pilhas (ou seja, o nó do nível 1 e os nós com dígitos diferentes de 1), armazena o topo da próxima pilha, isto é, o próximo nó com dígito diferente de 1. Para todos os outros nós o campo *next* armazena **null**.

A Figura 6.5 mostra como fica a estrutura, baseando-se nessa implementação, onde as flechas

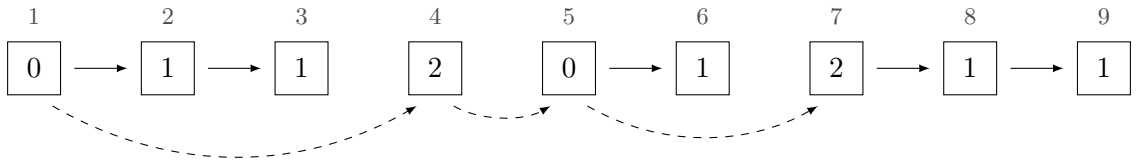

Figura 6.5

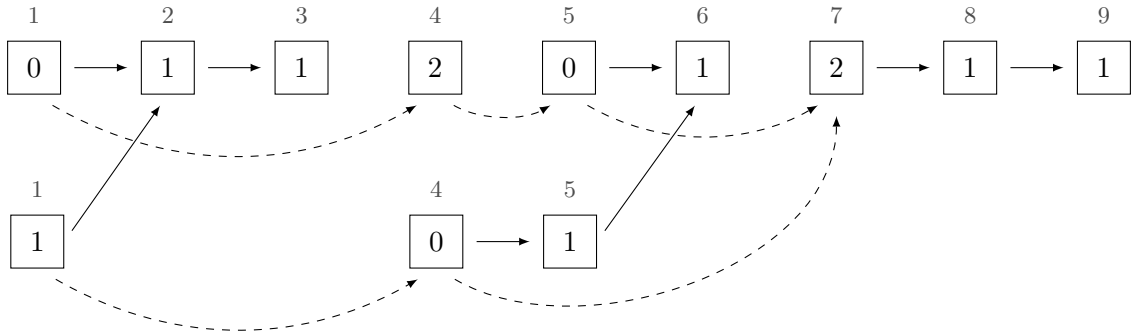

Figura 6.6

tracejadas indicam os ponteiros *next* não nulos, e a Figura 6.6 mostra a estrutura após as mesmas modificações que foram aplicadas na Figura 6.3. Note que, ao remover alguns ponteiros armazenados nos campos *child* (que apontam para o próximo nível), não é mais possível acessar nós de versões passadas. Contudo, a navegação pelos níveis fica um pouco mais complicada, já que um nó pode ter o campo *child* nulo e ainda assim existir um nó no próximo nível.

6.5 Representação

A representação de um nó da deque será similar à do Capítulo 5. Cada nó u terá os campos *prefix*, *suffix*, *child* e *next*.

Os campos *prefix* e *suffix*, como anteriormente, guardam um prefixo e um sufixo da deque, considerando u como raiz, mas agora estes campos são deques não persistentes de tamanho até 5. Usamos que {} cria uma deque não persistente vazia, e que cada deque tem funções homônimas às descritas no início do Capítulo 4, mas não recebem a deque e não devolvem uma nova deque, além disso, as operações Pop devolvem o valor removido, e uma função Size() devolve o número de elementos na deque não persistente. Por exemplo, se d é uma deque não persistente, d.PopBack() remove o último elemento de d e o devolve, modificando d.

Como discutido, o campo *next* aponta para o próximo nível com dígito diferente de 1, se existir e se u não tiver dígito 1, ou se u for o primeiro nível. O campo *child* aponta para o nó de nível $i+1$, se existir, e se este nó tem dígito 1.

6.6 Procedimento Fix detalhado

Vamos analisar mais detalhadamente os casos possíveis para o procedimento Fix. Ao aplicar o Fix em um nó de nível i com dígito 2, queremos transformar esse dígito em 0, possivelmente

modificando o nível $i+1$. Pela regularidade temos que o dígito do nível $i+1$ não é 2. A necessidade de tratar múltiplos casos se deve ao caso degenerado.

Para reduzir a descrição dos casos, vamos utilizar uma função FixDeques(l, r, L, R). Essa função recebe duas sub-deques (não persistentes) l e r, prefixo e sufixo do nível i e deixa ambas com 2 ou 3 elementos, possivelmente modificando as sub-deques L e R, prefixo e sufixo do nível $i+1$, tornando este nível vazio ou aumentando em no máximo 1 seu dígito. Este procedimento só funciona se existem mais de 3 elementos de nível i entre todas as deques da chamada, isto é, se $\text{Size}(l) + \text{Size}(r) + 2\,\text{Size}(L) + 2\,\text{Size}(R) > 3$.

Para aplicar um procedimento Fix, devemos primeiramente criar um nível $i+1$ vazio se este não existir. Após isso, trata-se os casos.

Caso 1. Se existem no máximo 3 elementos de nível i entre os níveis i e $i+1$, então adicione-os todos no prefixo do nível i, mantendo a ordem, e remova o nível $i+1$.

Caso 2. Caso contrário, sejam a e b os nós de nível i e $i+1$, respectivamente; execute a função FixDeques$(a.\textit{prefix}, a.\textit{suffix}, b.\textit{prefix}, b.\textit{suffix})$. Se o nível $i+1$ é agora o último nível e está vazio, remova-o.

Assumindo que a função FixDeques funciona, os casos tratam corretamente o Fix. No Caso 1, não existe nível $i+2$, pois o nível $i+1$ não tem mais que 1 elemento, então se não fosse o último nível, teria dígito 2, uma contradição pela semirregularidade da estrutura.

Além disso, o prefixo do nível i vai ter, ao final da função, 2 ou 3 elementos, já que é impossível o caso em que existe apenas 1 elemento de nível i entre os níveis i e $i+1$, pois isso implica que não existem elementos em $i+1$, e o nível $i+1$ teria dígito 2 (uma contradição pela semirregularidade da estrutura), ou não existe, e então o nível i é o último e tem exatamente 1 elemento, mas nesse caso o dígito desse nível não é 2.

Ainda resta, no procedimento Fix, arrumar os campos *child* e *next* dos nós modificados, que podem ter mudado. Essas dificuldades serão tratadas durante a implementação, discutida nas próximas seções.

6.7 Implementação de FixDeques

O Código 6.2 apresenta a implementação da função FixDeques(l, r, L, R), onde l e r são o prefixo e sufixo do nível i, que tem dígito 2, e L e R são o prefixo e sufixo do nível $i+1$, que não tem dígito 2. É possível que o nível $i+1$ não exista na deque atual; nesse caso L e R são deques não persistentes vazias.

O procedimento, para cada sub-deque, se esta tem pelo menos 4 elementos, remove 2 destes e os insere na sub-deque correspondente do nível $i+1$. Se a sub-deque tem no máximo 1 elemento, então um par é removido da sub-deque correspondente do nível $i+1$ e os dois elementos são adicionados na sub-deque. Dessa forma, ao final do procedimento, ambas as sub-deques têm 2 ou 3 elementos, e o dígito do nível i se torna 0. Também mostraremos que o dígito do nível $i+1$ aumenta em no máximo 1, se esse nível não for o último e estiver vazio.

Os **if**s das linhas 2 e 6 tratam os casos de remover elementos do nível i para adicioná-los ao nível $i+1$, e os **if**s das linhas 10 e 17 tratam os casos de remover um par do nível $i+1$ e adicioná-lo

Código 6.2: Função FixDeques

```
Require: l.Size() + r.Size() + 2 · L.Size() + 2 · R.Size() > 3
 1: function FixDeques(l, r, L, R)
 2:     if l.Size() ≥ 4 :
 3:         b = l.PopBack()
 4:         a = l.PopBack()
 5:         L.PushFront((a, b))
 6:     if r.Size() ≥ 4 :
 7:         a = r.PopFront()
 8:         b = r.PopFront()
 9:         R.PushBack((a, b))
10:     if l.Size() ≤ 1 :
11:         if L.Size() ≠ 0 :
12:             (a, b) = L.PopFront()
13:         else
14:             (a, b) = R.PopFront()
15:         l.PushBack(a)
16:         l.PushBack(b)
17:     if r.Size() ≤ 1 :
18:         if R.Size() ≠ 0 :
19:             (a, b) = R.PopBack()
20:         else
21:             (a, b) = L.PopBack()
22:         r.PushFront(b)
23:         r.PushFront(a)
```

ao nível i. Se alguma das sub-deques do nível $i+1$ é vazia, então este é o último nível (ou então teria dígito 2, uma contradição), logo, a sub-deque não vazia é o prefixo e sufixo deste nível. Então os **if**s das linhas 11 e 18 tratam corretamente esse caso. Dessa forma, o código faz como descrevemos, e funciona desde que as operações nas sub-deques L e R não sejam inválidas, ou seja, se nunca for executada uma operação de Push em uma sub-deque cheia ou uma de Pop quando o nível $i + 1$ está vazio.

As deques L e R não tem 5 elementos, pois se tivessem o dígito do nível $i + 1$ seria 2, não importando se este é o último nível ou não. Isso implica que as chamadas de Push nos **if**s das linhas 2 e 6 sempre funcionam. Além disso, no máximo um entre os **if**s das linhas 2 e 10 é executado, assim como entre os **if**s das linhas 6 e 17. Dessa forma, se algum dos primeiros dois **if**s é executado, pela ordem que as operações são feitas temos que o nível $i + 1$ não está vazio e no máximo um dos últimos dois **if**s também é executado, logo as operações de Pop não serão executadas com o nível $i + 1$ vazio nesse caso.

Os únicos casos em que uma operação de Pop podem ser inválidas são:

- Os últimos dois **if**s são executados, ou seja, se l e r tem no máximo um elemento; e o nível $i+1$ tem no máximo um par. Como a função FixDeques só é chamada se existem mais que 3 elementos de nível i entre todas as deques, o único caso que isso pode acontecer é quando as deques l e r tem exatamente um elemento, e existe um par em uma das deques L ou R. Mas nesse caso o dígito do nível i é 1, e FixDeques não é chamada.

- Apenas um dos últimos dois **if**s é executado, e o nível $i+1$ é vazio, ou seja, o nível i é o último. Isso ocorre se uma entre l e r tem 2 ou 3 elementos, e a outra tem no máximo 1, mas este caso não ocorre pois o dígito do nível i não seria 2.

Logo, não existe situação em que a execução da função FixDeques faz chamadas de Push ou Pop inválidas, e portanto o Código 6.2 funciona corretamente.

6.8 Implementação de Fix

O Código 6.3 apresenta a implementação de Fix(a), uma função que recebe um nó de nível i com dígito 2 e aplica modificações nele que transformam seu dígito em 0, aumentando em até 1 o dígito do nível $i+1$. O nó a é modificado, mas uma cópia do nível $i+1$ é feita, então não modificamos o nó original de nível $i+1$. Para não modificar um nó da estrutura, é necessário chamar a função com uma cópia do nó de nível i. Usamos a função Copy(x) para devolver uma cópia de um nó x, em tempo constante. Se x for **null**, devolve-se um nó vazio (com as sub-deques vazias e os campos nulos).

A função Digit($a, last$) devolve o dígito do nó a, onde *last* é **true** se não existe nível depois de a, e **false** caso contrário. Receber *last* é necessário pois um nó pode ter o campo *child* nulo, mas ainda assim existir um nó no próximo nível. Nesse caso, o nó de próximo nível é o campo *next* da cabeça da sub-pilha que contém a. Veja a Figura 6.5, o nó de nível 3 tem campo *child* nulo, mas existe nó de nível 4, e um ponteiro para este está armazenado apenas no nó de nível 1.

As linhas 10-17 copiam o nó de nível $i+1$ na variável b, criando um nó vazio se este não existir. Este nó pode estar tanto no campo *child* quanto no campo *next* de a, se ele existir tiver dígito 1 ou 0, respectivamente. Também é armazenado na variável *last* o valor **true** se não existe nível $i+2$, e **false** caso contrário. A notação de Iverson [`predicado`] devolve **true** (ou 1) se `predicado` é verdade e **false** (ou 0) se é falso.

As linhas 19-30 resolvem o Caso 1, como discutido na Seção 6.6, adicionando todos elementos das sub-deques dos níveis i e $i+1$ no prefixo do nó a, e apagando b. Usamos o fato que cada sub-deque em ambos os níveis tem no máximo 1 elemento.

O Caso 2 é tratado nas linhas 31-34. Na Seção 6.7 discutimos que a função FixDeques está correto. O **if** da linha 33 verifica se não existe nível $i+2$ e ambas as sub-deques do nível $i+1$ estão vazias. Nesse caso o nível $i+1$ é removido.

Portanto, os casos são tratados corretamente. As linhas seguintes arrumam os campos *child* e *next* de a e b. Se b existe e tem dígito 1, este deve ser armazenado no campo *child* de a (linha 39). Se o nível $i+1$ tinha dígito diferente de 1 (esse dígito era 0 pela semirregularidade da estrutura), então esse nível era o *next* de a, e precisamos mudar esse valor para ser o próximo nível com dígito diferente de 1 (linhas 36-38). Usamos que se b tinha dígito diferente de 1, então $a.child$ era nulo.

Se b não tem dígito 1, ou não existe, deve ser armazenado no campo *next* de a (linhas 43-44). Se b tinha dígito 1 no início da operação, e ainda existe, é necessário guardar o valor antigo de $a.next$ em $b.next$ (linhas 41-42). Caso contrário, o campo $b.next$ permanece o mesmo, pois já apontava para o próximo nível com dígito diferente de 1.

Note que é conveniente armazenar o próximo dígito diferente de 1, e não apenas o próximo dígito 2, pois assim ao final do Fix temos garantia que a continua sendo um topo de pilha, pois tem

Código 6.3: Operação FIX.

```
1: function DIGIT(a, last)
2:     digit = [2, 1, 0, 0, 1, 2]          ▷ digit[i + 1] = dígito de uma sub-deque com i elementos
3:     if a.prefix.SIZE() = 0 and last :
4:         return digit[a.suffix.SIZE() + 1]
5:     else if a.suffix.SIZE() = 0 and last :
6:         return digit[a.prefix.SIZE() + 1]
7:     else
8:         return max(digit[a.prefix.SIZE() + 1], digit[a.suffix.SIZE() + 1])
Require: a é de nível i, o primeiro nó com dígito 2.
9: function FIX(a)
10:    if a.child ≠ null :
11:        b = COPY(a.child)
12:        last = [a.next = null and b.child = null]
13:    else if a.next ≠ null :
14:        b = COPY(a.next)
15:        last = [b.next = null and b.child = null]
16:    else
17:        b = new NODE({}, null, {})
18:        last = true
19:    if a.prefix.SIZE() + a.suffix.SIZE() + 2 b.prefix.SIZE() + 2 b.suffix.SIZE() ≤ 3 :    ▷ Caso 1
20:        if b.prefix.SIZE() ≠ 0 :
21:            (x, y) = b.prefix.POPFRONT()
22:            a.prefix.PUSHBACK(x)
23:            a.prefix.PUSHBACK(y)
24:        if b.suffix.SIZE() ≠ 0 :
25:            (x, y) = b.suffix.POPFRONT()
26:            a.prefix.PUSHBACK(x)
27:            a.prefix.PUSHBACK(y)
28:        if a.suffix.SIZE() ≠ 0 :
29:            a.prefix.PUSHBACK(a.suffix.POPFRONT())
30:        b = null
31:    else                                                                    ▷ Caso 2
32:        FIXDEQUES(a.prefix, a.suffix, b.prefix, b.suffix)
33:        if b.prefix.SIZE() = 0 and b.suffix.SIZE() = 0 and last :
34:            b = null
35:    if b ≠ null and DIGIT(b, last) = 1 :
36:        if a.child = null :                    ▷ Nível i + 1 tinha dígito diferente de 1.
37:            a.next = b.next
38:            b.next = null
39:        a.child = b
40:    else
41:        if b ≠ null and a.child ≠ null :
42:            b.next = a.next
43:        a.next = b
44:        a.child = null
```

dígito 0.

6.9 Implementação das operações

Com a função Fix implementada, é fácil implementar as operações da deque com consumo de tempo constante. Pela regularidade da estrutura, é sempre possível realizar um Push ou Pop no primeiro nível, e após isto basta chamar a função Fix no nó adequado, como discutido na Seção 6.3.

Código 6.4: Operações da deque

```
1:  function Check(a)
2:      if Digit(a, [a.child = null and a.next = null]) = 2 :
3:          Fix(a)
4:      else if a.next ≠ null and Digit(a.next, [a.next.child = null and a.next.next = null]) = 2 :
5:          a.next = Copy(a.next)
6:          Fix(a.next)
7:  function PushFront(a, x)
8:      a = Copy(a)
9:      a.prefix.PushFront(x)
10:     Check(a)
11:     return a
12: function PopFront(a)
13:     a = Copy(a)
14:     if a.prefix.Size() ≠ 0 :
15:         x = a.prefix.PopFront()
16:     else
17:         x = a.suffix.PopFront()
18:     Check(a)
19:     return (x, a)
20: function Front(a)
21:     if a.prefix.Size() ≠ 0 :
22:         return a.prefix.Front()
23:     else
24:         return a.suffix.Front()
```

O Código 6.4 mostra as implementações das versões Front das operações da deque. A função Check(a) realiza um Fix na deque com topo a, se necessário, ou seja, se esta não é regular. Logo, após essa operação, é garantido que a estrutura é regular, como discutido na Seção 6.3.

Note que as operações PushFront e PopFront copiam o topo da deque, e a função Check copia o campo *next* deste elemento se for necessário modificá-lo. A função Fix(a) modifica a, mas está na verdade recebendo uma cópia do nó. Como a implementação é feita de acordo com a discussão na Seção 6.4, garantimos que todo nó copiado é apenas apontado por outros nós copiados, e assim a implementação é funcional, e portanto persistente.

Os **if**s das operações PopFront e Front checam os casos degenerados, já que, como a estrutura é regular, se uma das sub-deques de a é vazia então a é tanto o topo quanto o último nível da estrutura.

A operação k-th(d, k) não foi implementada, mas pode ser feita de forma similar, embora um pouco mais complicada, ao Código 5.3. A Tabela 6.1 mostra o consumo de tempo e espaço da

	Tempo/Espaço
Deque()	$\mathcal{O}(1)/\mathcal{O}(1)$
PushFront(q, x)	$\mathcal{O}(1)/\mathcal{O}(1)$
PushBack(q, x)	$\mathcal{O}(1)/\mathcal{O}(1)$
Front(q)	$\mathcal{O}(1)$
Back(q)	$\mathcal{O}(1)$
PopFront(q)	$\mathcal{O}(1)/\mathcal{O}(1)$
PopBack(q)	$\mathcal{O}(1)/\mathcal{O}(1)$
k-th(q, k)	$\mathcal{O}(\lg n)$

Tabela 6.1: Consumo de tempo e espaço da implementação deste capítulo, onde n é o tamanho da deque.

	Representação binária	Skew-binary	Recursiva	Kaplan e Tarjan
Deque()	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$
PushFront(q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
PushBack(q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
Front(q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(1)$
Back(q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(1)$
PopFront(q)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
PopBack(q)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
k-th(q, k)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 6.2: Comparação do consumo de tempo e espaço das deques implementadas nos Capítulos 4, 5 e 6, onde n é o tamanho da deque.

implementação discutida neste capítulo. A Tabela 6.2 compara o consume de tempo e espaço de todas as deques persistentes apresentadas neste trabalho.

Capítulo 7

Técnicas gerais

Nos capítulos anteriores, discutimos como tornar persistentes estruturas de dados específicas (pilhas, filas, deques). Como dito na introdução, outro caminho é criar técnicas gerais para tornar qualquer ED persistente.

7.1 Modelo de computação

Para tomar uma abordagem tão geral, é necessário definir formalmente o conceito de estrutura de dados. Usaremos o modelo de *estruturas ligadas*, no qual uma estrutura de dados é um conjunto de *nós*, ou *objetos*, cada um contendo um número constante de campos. Cada campo pode armazenar um valor (um inteiro ou booleano, por exemplo) ou um ponteiro para outro nó. Ponteiros também podem armazenar o valor especial **null**. Acesso à estrutura é dado por um número constante de *nós de entrada*.

Uma pilha implementada com lista ligada, como no Capítulo 3, é uma estrutura desse tipo, onde o primeiro elemento da lista é o nó de entrada. A estrutura recursiva para deques do Capítulo 5 também é dessa forma, e tem nós de dois tipos: os nós da estrutura e nós que são os pares. O nó de entrada também é o primeiro nó da estrutura. Uma árvore de busca binária também é deste tipo, já que cada nó tem dois ponteiros (para o filho esquerdo e direito), e um valor; e a raiz é o nó de entrada. A estrutura para resolver o problema do Ancestral de Nível no Capítulo 1 não é deste tipo, pois o pré-processamento armazena em cada nó um vetor de tamanho $\lfloor \lg d \rfloor$, onde d é a altura do nó.

Vamos formalizar também as definições de operações de acesso e modificação. Uma *operação de acesso* produz um *conjunto de acessos*. No início da operação, este conjunto está vazio. A cada passo, chamado de *passo de acesso*, adiciona-se um nó a este conjunto. O nó adicionado deve ser um nó de entrada ou deve ser indicado por um ponteiro em algum nó já no conjunto de acessos. O tempo consumido por uma operação de acesso é o número de passos de acesso. Operações também devolvem valores, calculados usando seu conjunto de acessos.

A busca em uma ABB por um elemento é um exemplo de operação de acesso, que devolve se o elemento está ou não na ABB. Um exemplo mais complicado é a operação K-TH da deque recursiva, que primeiro encontra o nó da estrutura em que se localiza o k-ésimo elemento, e depois "descasca" os pares para encontrar e devolver este elemento.

Uma *operação de modificação* é parecida com uma de acesso. Ela consiste de passos de acesso, que funcionam da mesma forma, e *passos de modificação*. Em um passo de modificação, uma das seguintes operações é feita:

- Cria-se um novo nó, e este é adicionado ao conjunto de acessos.
- Modifica-se um campo de algum nó do conjunto de acessos. Se um ponteiro foi modificado, seu novo valor deve ser **null** ou deve ser um dos nós do conjunto de acessos.
- Troca-se um nó de entrada. Seu novo valor deve ser **null** ou um nó do conjunto de acessos.

O tempo consumido por esta operação de modificação é o número total de passos, e o tempo de modificação é o número de passos de modificação.

Adicionar um elemento em uma ABB, por exemplo, consiste de achar sua posição, que pode levar até h passos de acesso, onde h é a altura da árvore, e criar um novo vértice ali, modificando um dos ponteiros desse nó. Logo esta operação consome tempo $\mathcal{O}(h)$ e tempo de modificação $\mathcal{O}(1)$.

Essas definições descrevem uma estrutura de dados efêmera, ou seja, uma ED na qual operações de acesso e modificação podem apenas ser feitas na versão mais atual da estrutura. Nosso objetivo é tornar esta estrutura genérica em uma estrutura total ou parcialmente persistente.

Consideramos que o número total de operações de modificação é m, e o número de operações de acesso é a. Da mesma forma, o número total de passos de modificação é M e de passos de acesso é A. A i-ésima operação de modificação cria a versão i da estrutura. A versão 0 é a estrutura vazia. Se estamos considerando uma estrutura totalmente persistente, temos uma árvore de versões, caso contrário, uma lista.

Consideramos que cada operação recebe também, além de seus parâmetros normais (por exemplo, um valor para ser encontrado, no caso de uma busca em ABB) , um inteiro que indica sob qual versão deve ser feita essa operação. Dizemos que $i \to j$ se a operação que cria a versão j é feita sobre a versão i.

Note que, apesar de estarmos tratando apenas de estruturas ligadas, a versão persistente desta estrutura pode não ser ligada. Por exemplo, um dos métodos poderia armazenar um vetor de tamanho variável em cada nó, logo a estrutura deixaria de ser ligada.

7.2 Offline

Em geral, estamos interessados em implementações online das estruturas persistentes, ou seja, uma operação deve ser completada para se ter acesso à próxima, porém, vamos considerar o caso offline, em que todas as $m+a$ operações são conhecidas de antemão. Obteremos uma implementação totalmente persistente que não faz nenhuma suposição adicional sobre a estrutura de dados, não aumenta assintoticamente o tempo consumido pelas operações, e gasta apenas $\mathcal{O}(1)$ de espaço adicional por passo de modificação, que é o melhor que pode-se esperar. A versão original desta solução foi dada por meu amigo, Arthur Nascimento, enquanto discutíamos sobre como implementar uma deque persistente.

Inicialmente, constrói-se a árvore de versões. O vértice v_i representa a versão i. Neste deve-se armazenar, além da lista de adjacência (v_k tal que $i \to k$), a operação de modificação que criou a

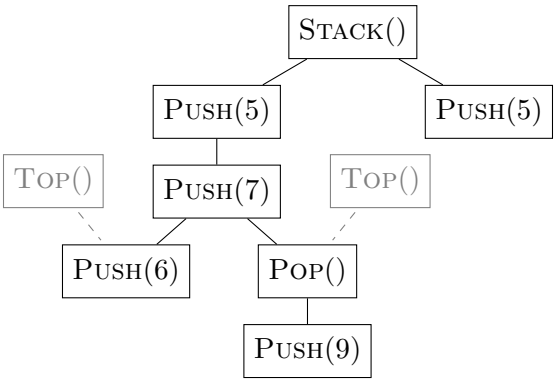

Figura 7.1: Árvore de versões do Exemplo 3.1, com argumentos para fazer cada operação de modificação indicados. Os nós mais claros (cinzentos) indicam as operações de acesso, conectadas à versão na qual são realizadas.

versão i (seus parâmetros), e uma lista das operações de acesso que são feitas na versão i. Isso pode ser feito em tempo e espaço $\mathcal{O}(m + a)$, ou seja, proporcional ao número de operações. A Figura 7.1 mostra a árvore de versões do exemplo de pilha do Capítulo 3.

Após montar a árvore de versões, realizamos uma busca em profundidade (DFS) nesta árvore, seguindo a seguinte ideia:

1. Ao acessar um vértice v_i pela DFS, a estrutura está na versão j, onde $j \to i$.
2. Aplicamos a operação, transformando a versão j em i.
3. Calculamos o valor de retorno para as operações de acesso feitas na versão i e em todas as versões descendentes de i recursivamente.
4. Ao final do acesso ao vértice v_i, revertemos as mudanças feitas, como explicado adiante, e a estrutura está novamente na versão j.

Dessa forma, no passo 3, podemos acessar os nós que são filhos de v_i recursivamente, já que, devido ao passo 4, quando o acesso a eles terminar, a versão da estrutura será a versão i, e no passo 4 de v_i basta reverter as mudanças feitas nessa versão.

Para reverter mudanças de forma genérica, sempre que um passo de modificação altera um campo de um objeto, guardamos uma tripla $(obj, field, oldValue)$ indicando que o campo *field* do objeto *obj* mudou de valor, e seu valor anterior era *oldValue*. Isso é informação o bastante para, ao final do procedimento, reverter essa mudança. Se o passo de modificação altera um nó de entrada, este também pode ser armazenado desta forma, por exemplo, usando um objeto para armazenar os nós de entrada.

Como não impusemos restrições nas operações de mudança, elas podem modificar um mesmo campo muitas vezes. Logo, é necessário que o valor original seja restaurado, e não um valor intermediário. Para isso, desfazemos as mudanças na ordem inversa que foram feitas, usando uma pilha.

O Código 7.1 mostra uma implementação, em alto nível, do algoritmo discutido. Assumimos que, ao realizar uma operação, seu valor de retorno seja armazenado em algum lugar. A operação na linha 4 deve iterar pela lista de operações de acesso do vértice v_i e aplicá-las. Como assumimos

Código 7.1: Persistência total offline

```
1: function DFS(v_i)
2:     mod = STACK()
3:     Aplicar a i-ésima operação de modificação, guardando valores antigos em mod.
4:     Responder todas as operações de acesso realizadas na versão i.
5:     for v_k ∈ lista de adjacência de v_i :
6:         DFS(v_k)
7:     while mod.SIZE() > 0 :
8:         (obj, field, oldValue) = mod.POP()
9:         obj.field = oldValue
10: function TOTALPERSISTENCEOFFLINE()
11:     Montar a árvore de versões, com raiz v_0.
12:     DFS(v_0)
```

que a ED começa na versão 0, temos que o vértice v_0 é a raiz. A linha 3 não faz nada quando o vértice é v_0.

Proposição 7.1. *Se a função* DFS(v_i) *é chamada com a versão j da estrutura de dados, onde $j \to i$, então ela aplica corretamente todas as operações de acesso de alguma versão cujo vértice é descendente de v_i. Além disso, ao final da execução da função, a versão da estrutura é j.*

Demonstração. A prova é por indução na altura de v_i. No início da função, estamos na versão j, então ao aplicar a operação i, a estrutura muda para a versão i. Além disso, todas as mudanças para reverter a versão i para j estão armazenadas em *mod*.

A linha 4 aplica as operações de acesso da versão i. O **for** da linha 5 chama, para cada filho de v_i, a função DFS neste. Seja v_x o primeiro filho acessado; pela hipótese de indução, como v_x tem altura menor que v_i e a estrutura está na versão i e $i \to x$, temos que a função aplica corretamente as operações de acesso das versões na subárvore de v_x, e ao final da chamada de DFS(v_x), a estrutura está novamente no estado i.

Dessa forma, as chamadas da função DFS para todos os filhos de v_i funcionam corretamente, e por isso as operações de acesso de todos os descendentes de v_i são realizadas. Por último, o **while** da linha 7 desfaz as mudanças feitas na versão i e devolve a estrutura para a versão j, completando a prova. □

Então, é possível transformar qualquer estrutura ligada em totalmente persistente, de forma offline, com aumento no consumo de tempo de $\mathcal{O}(M + A)$ e de espaço $\mathcal{O}(M + a)$.

7.3 Implementação funcional

A técnica que usamos nas estruturas apresentadas até agora é tornar a implementação funcional, ou seja, nunca modificar nenhum campo de um nó já criado. Esse tipo de implementação é bastante ligada a linguagens funcionais, e implementações desta forma vão de pilhas a ABBBs [6, 8, 10, 11].

Se for possível implementar a estrutura desta forma, ela se torna totalmente persistente. De fato, se temos os nós de entrada da versão i, e nenhum nó foi modificado, os nós que podem ser acessados a partir desses são exatamente os mesmos de quando a versão i foi criada, independente

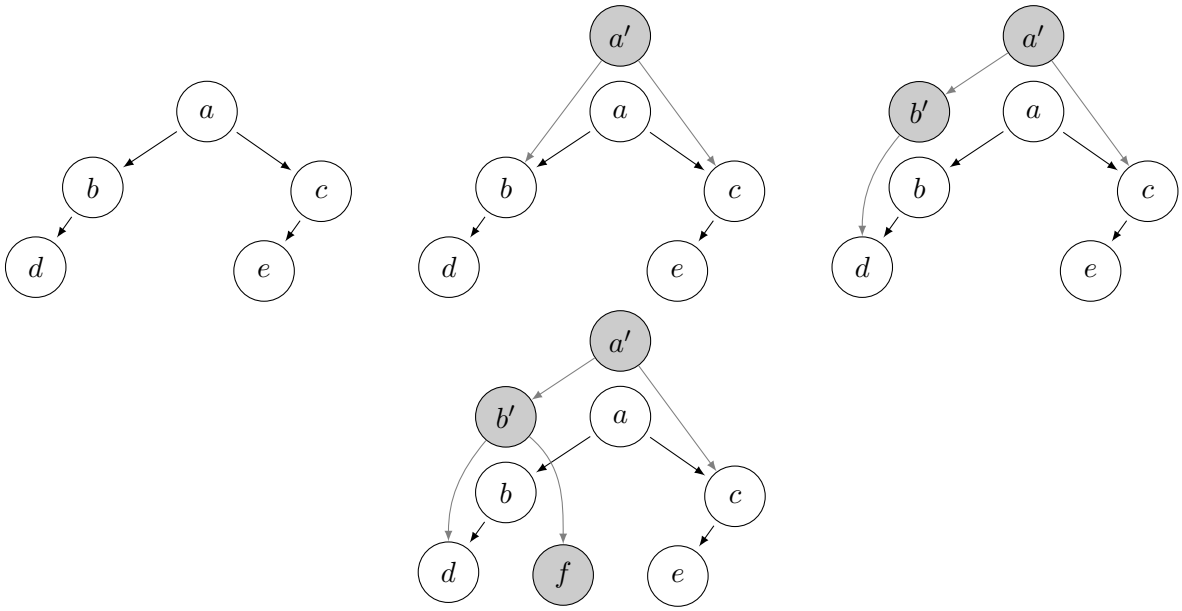

Figura 7.2: Adição de um nó f à direita de b com uma implementação totalmente funcional. Em cinza, os nós criados ou copiados durante a adição.

de versões criadas posteriormente. Logo, apenas armazenando os nós de entrada (e não apagando nenhum nó), tornamos nossa estrutura totalmente persistente.

Algumas estruturas ligadas já funcionam naturalmente dessa forma, como por exemplo a pilha implementada no Capítulo 3, onde nunca é necessário alterar nenhum nó. Quando a estrutura não funciona dessa forma, é possível tentar "forçar" isso, copiando os nós que seriam modificados (e seus ancestrais). A estrutura recursiva de deque do Capítulo 5 faz isso, copiando todos os nós acessados durante um PUSH ou POP.

Para fazer isso em uma estrutura qualquer, supomos que, sempre que adicionamos um nó u ao conjunto de acessos, todos seus ancestrais já estão neste conjunto. Ao adicionar esse nó, na verdade adicionamos uma cópia u' dele, e para todo nó v no conjunto de acessos que tem um ponteiro para u, trocamos este ponteiro por u'.

Note que nós no conjunto de acessos podem ser modificados, mas o importante é que nenhum nó criado em uma versão anterior é modificado, logo todas essas versões continuam funcionando.

A Figura 7.2 mostra cada passo de uma operação de inserção em uma ABB usando esse método. Os nós que existiam no começo da operação não foram modificados, então a árvore "visível" a partir de a continua a mesma. Note que alguns nós são visíveis nas duas versões, como o d e o e, já que não foram modificados (nem nenhum de seus descendentes). Se quiséssemos armazenar também ponteiros de pai em cada nó, uma implementação dessa forma não seria possível.

Supusemos que, ao adicionar um nó ao conjunto de acessos, todos seus ancestrais já estavam neste conjunto (na verdade, suas cópias), portanto não existe caminho de um nó não copiado para um nó que tem uma cópia no conjunto de acessos. Assim, todos os nós atingíveis a partir da nova versão ou são as cópias criadas ou são nós que nunca foram adicionados ao conjunto de acessos, ou seja, são exatamente os nós que seriam atingíveis se tivéssemos copiado a estrutura inteira. Logo, a estrutura ligada funciona e é totalmente persistente. Note que na Figura 7.2 não é possível acessar, a partir de a', a versão anterior de um nó copiado (os nós a e b).

Sem essa suposição, poderia ser possível alcançar a versão antiga de um nó que já foi copiado. Este foi o problema encontrado na primeira ideia de implementação da deque de Kaplan e Tarjan, como discutido na Seção 6.4.

Note que, se os ponteiros da estrutura formam uma floresta direcionada, onde os nós de entrada são as raízes e as arestas vão em direção contrária às raízes, cada nó tem exatamente um caminho até ele, logo a suposição sobre adicionar um nó ao conjunto de acessos é sempre satisfeita, não importa a ordem na qual são feitas as adições ao conjunto de acessos. A estrutura discutida no Capítulo 5 é uma árvore direcionada, e na Seção 6.4 a implementação inicial discutida para a estrutura do Capítulo 6 é modificada para também se tornar uma árvore direcionada.

Assumindo que, ao adicionar um nó ao conjunto de acessos, as mudanças de ponteiros possam ser feitas em tempo constante (trivial em uma floresta direcionada), se a estrutura segue as restrições discutidas, esse método torna a estrutura totalmente persistente com aumento no consumo de tempo de $\mathcal{O}(M+A)$ e espaço de $\mathcal{O}(M+A_m)$, onde A_m é o número de passos de acesso que ocorrem durante operações de modificação, já que é necessário copiar o nó mesmo que este não seja modificado.

7.4 Fat node

É possível tornar uma estrutura ligada parcialmente persistente aumentando os seus nós para armazenar todas as mudanças já feitas. Essa técnica foi inicialmente apresentada por Driscoll, Sarnak, Sleator e Tarjan [5].

Modifica-se cada nó da estrutura para que cada campo seja substituído por um vetor de pares, que guarda pares (i, v), indicando que a i-ésima operação de modificação alterou aquele campo para o valor v. Em vez de modificar um campo de um objeto, adiciona-se um par no vetor correspondente, indicando a versão atual e o novo valor para o campo. Como estamos fazendo uma implementação parcialmente persistente, o índice de versão adicionado é sempre maior que todos os anteriores, dessa forma o vetor se mantém ordenado (considerando apenas os índices).

Para realizar um acesso ao campo c na versão i, é necessário procurar no vetor associado a c o par com maior índice que não excede i, ou seja, a última atualização feita até no máximo a versão i. Isso pode ser feito com busca binária em tempo logarítmico no tamanho do vetor, já que a informação é armazenada em ordem. Diferente das outras técnicas, esta também aumenta o tempo de um passo de acesso.

Utilizar esta técnica transforma qualquer estrutura ligada em uma estrutura parcialmente persistente, com aumento no consumo de tempo de $\mathcal{O}((M + A) \lg M)$ e espaço de $\mathcal{O}(M)$.

7.5 Node copying

No método Fat node, ao armazenar todas as modificações de um campo, precisamos de tempo logarítmico para descobrir o valor desse campo em uma dada versão. O método *Node copying* também consegue persistência parcial, mas diminui esse tempo, armazenando apenas um número constante de modificações em um mesmo nó, e fazendo uma cópia do nó quando este ficar muito grande. Este método também foi apresentado por Driscoll et al. [5].

Precisamos, contudo, fazer mais suposições sobre a estrutura. Supomos que os nós da estrutura

ligada têm grau de entrada constante, ou seja, existe uma constante *in* tal que, a qualquer momento, para qualquer sequência de operações, vale que, para qualquer nó u, o número de campos que tem ponteiros que apontam para u, entre todos os nós da estrutura, é no máximo *in*.

Cada nó deve armazenar, além de seus campos usuais, um vetor de triplas *changes* de tamanho *in* que armazena mudanças feitas nos seus campos, ou seja, se este vetor tem a tripla $(version, field, value)$ então na versão *version* o campo *field* foi modificado para o valor *value*. Esse vetor é inicialmente vazio, e armazena mudanças feitas ao nó depois de sua criação.

O método Node copying copia um nó u sempre que este é modificado em um passo de modificação. Após isso, é necessário modificar os ponteiros que apontam para u. Para evitar ter que copiar todos os ancestrais de u, guardamos estas modificações de ponteiros em seus campos *changes*, se estes tiverem espaço.

Se algum dos nós que apontam para u não tiver espaço livre em seu campo *changes*, uma cópia deste nó deve ser feita, armazenando a versão mais nova de cada campo (e deixando o vetor *changes* vazio, no novo nó). Além disso, os nós que apontam para este devem ser modificados, o que pode gerar mais cópias.

Adicionamos um campo $\mathcal{T}$ a cada nó da estrutura, que é o índice da versão em que o nó foi criado, e um campo *copy* que é um ponteiro para a cópia criada diretamente a partir deste nó, ou **null** se não existir. Quando uma cópia é criada, a versão antiga do nó nunca mais é modificada, logo nenhuma outra cópia é criada diretamente a partir deste nó. Dessa forma, cada nó u representa o nó correspondente da estrutura no intervalo de versões $[u.\mathcal{T}, u.copy.\mathcal{T} - 1]$ se este tiver cópia, caso contrário o nó representa a versão atual do nó correspondente na estrutura, e é chamado de *nó ativo*. Como estamos tratando de persistência parcial, apenas nós ativos são modificados.

Implementações

Como estamos lidando com estruturas de dados gerais, não apresentaremos pseudocódigo para as operações de acesso e modificação, que podem variar para cada estrutura, mas sim para uma interface que estas operações usam para acessar e modificar a estrutura.

Assumimos que um vetor *entry* armazena todas as versões de nós de entrada. Dessa forma, operações de acesso podem acessar nós de entrada de qualquer versão em tempo constante. Além disso, assumimos que uma variável *current* armazena o número da versão atual da estrutura. No começo de toda operação de modificação esta variável deve ser incrementada, e os nós de entrada da versão anterior devem ser copiados para a nova versão.

Na Subseção **Acesso**, apresentaremos a função de interface ACCESS, que deve ser usada quando for necessário acessar um campo de um nó da estrutura, isto é, as operações de acesso devem usar ACCESS$(u, field, i)$ para acessar o campo *field* de u na versão i, em vez de acessar o campo diretamente, como em $u.field$.

Na Subseção **Modificação**, apresentaremos uma versão de ACCESS para ser usada durante operações de modificação, e também a função de interface MODIFY$(u, field, value)$, que deve ser usada para modificar o campo *field* de u para o valor *value*, em vez de fazer isso diretamente, como em $u.field = value$. As outras funções desenvolvidas nessa subseção são auxiliares, usadas direta ou indiretamente por ACCESS e MODIFY. Para deixar isto claro, os nomes das funções de interface são SUBLINHADAS.

Acesso

O acesso a um campo de um nó em uma dada versão pode ser feito como no Código 7.2. Note que estamos assumindo que as modificações estão armazenadas no vetor *changes* na ordem em que foram feitas.

Código 7.2: Acesso a um campo durante uma operação de acesso.

```
Require: u representa a versão version.
1: function ACCESS(u, field, version)
2:     value = u.field
3:     for (version', field', value') ∈ u.changes :
4:         if field = field' and version ≥ version' :
5:             value = value'
6:     return value
```

Para devolver o valor do campo na versão *version*, aplicamos todas as modificações realizadas a este campo em versões que são no máximo *version*. Como o vetor *entry* armazena os nós de entrada de cada versão, com a função de interface ACCESS é possível realizar uma operação de acesso em qualquer versão da estrutura com aumento constante no tempo por passo de acesso.

Modificação

Em uma operação de modificação, modificar um nó do conjunto de acessos pode levar à cópia de outros nós que já estão no conjunto de acessos. Para evitar que isso nos traga problemas, vamos garantir que no máximo uma cópia de cada nó seja feita. Dessa forma, quando um passo de modificação da estrutura tentar acessar ou modificar um campo do nó u, ou u é ativo ou $u.copy$ existe e foi criado nesta versão.

A função ACCESS$(u, field)$ do Código 7.3 funciona de forma parecida com a função ACCESS$(u, field, version)$ do Código 7.2, mas sempre acessa a versão mais recente (*current*) do campo. Esta função não é usada em nenhuma das funções apresentadas no resto da seção, mas, como discutido, deve ser usada pelos *passos de acesso* durante uma *operação de modificação* da estrutura de dados. A função ACTIVE(u) é usada como função auxiliar em muitas funções apresentadas nesta seção, e devolve a versão ativa do nó u (ou seja, u ou $u.copy$).

Código 7.3: Acesso a um campo durante uma operação de modificação.

```
Require: u é ativo ou sua cópia foi criada nessa versão.
1: function ACCESS(u, field)
2:     return ACCESS(ACTIVE(u), field, current)
Require: u é ativo ou sua cópia foi criada nessa versão.
3: function ACTIVE(u)
4:     if u = null or u.copy = null :
5:         return u
6:     else
7:         return u.copy
```

Para poder atualizar os ponteiros de nós ativos que apontam para um nó u, armazenamos em u um vetor *parents*, de *in* posições, que armazena quais nós ativos apontam para u. Se u não é ativo,

esse vetor tem apenas **null**s. Dizemos que *parents* armazena *ponteiros de volta*.

Quando um campo de ponteiro de um nó x é alterado de y para z, é necessário atualizar o vetor *parents* de y e z, removendo x do vetor de y e adicionando-o ao vetor de z. A função ChangeParent(u, a, b) modifica o vetor $u.parents$, trocando uma ocorrência de a por b. A função ChangePointer$(u, field, value)$ muda o ponteiro $u.field$ para $value$ (usando ChangeParent, se necessário); essa função assume que o nó u foi criado nesta versão, portanto podemos mudar diretamente seus campos, sem ser necessário adicionar mudanças ao vetor *changes*. O Código 7.4 mostra a implementação destas duas funções.

Código 7.4: Implementação de ChangeParent e ChangePointer.

```
1:  function ChangeParent(u, a, b)
2:      for i = 1 to in :
3:          if u.parents[i] = a :
4:              u.parents[i] = b
5:              break
Require: u é um nó criado nesta versão.
Require: field é um campo de ponteiro.
6:  function ChangePointer(u, field, value)
7:      if u.field ≠ null :
8:          ChangeParent(u.field, u, null)
9:      u.field = Active(value)                    ▷ value pode já ter sido copiado.
10:     if u.field ≠ null :
11:         ChangeParent(u.field, null, u)
```

A função ChangeParent(u, a, b) funciona se o vetor *parents* está atualizado pois, ao trocar a por b, se $a \neq$ **null**, então a apontava para u, logo estava em *parents*; se $a =$ **null**, então, como *in* é o grau de entrada máximo de um nó, pelo menos uma posição do vetor tem o valor **null**.

Código 7.5: Modificação feita por um passo de modificação.

```
Require: u é ativo ou sua cópia foi criada na versão atual (current).
1:  function Modify(u, field, value)
2:      u = Active(u)
3:      if u.version < current :
4:          u.copy = Copy(u)
5:          u = u.copy
6:      if field é campo de ponteiro :
7:          ChangePointer(u, field, value)
8:      else
9:          u.field = value
```

A função Modify$(u, field, value)$ modifica o campo *field* de u, e deve ser chamada em passos de modificação da operação com índice *current*. A função cria uma cópia do nó u, se esta não existe e u não foi criado nessa versão (usando a função Copy), e depois modifica o campo *field* diretamente, utilizando ChangePointer se necessário. O Código 7.5 mostra a implementação da função Modify. O nó u é um nó do conjunto de acessos e, como discutido anteriormente, um destes casos ocorre:

- u tem uma cópia — A chamada a Active na linha 2 troca u por esta cópia; ou

- u não tem cópia e não foi criado nesta versão — O **if** da linha 3 cria uma cópia deste nó e troca u por esta cópia; ou
- u foi criado nessa versão — u não é modificado.

Após qualquer um destes casos, u é um nó que foi criado nesta versão, e então as linhas 6-9 mudam o campo *field*. Resta detalhar a função Copy, que cria a cópia de um nó e modifica os ponteiros em nós ativos que apontam para ele.

Código 7.6: Cópia de um nó na versão *current*, atualizando os ponteiros que apontam para ele.

```
1:  function Copy(u)
2:      u' = RawCopy(u)                    ▷ A função RawCopy(u) copia todos os campos de u.
3:      u'.changes = {}                    ▷ Inicializando changes com vetor vazio.
4:      u'.T = current
5:      for (version', field', value') ∈ u.changes :
6:          u'.field' = value'
7:      if u é nó de entrada :
8:          Trocar u por u' na posição correta do vetor entry.
9:      for pointer ∈ campos de ponteiros :
10:         if u'.pointer ≠ null :
11:             ChangeParent(u'.pointer, u, u')
12:     u.parents = vetor apenas com nulls
13:     for i = 1 to |u'.parents| :
14:         v = u'.parents[i]
15:         if v ≠ null :
16:             field = campo de v tal que Access(v, field) = u.
17:             if v.T = current :
18:                 v.field = u'
19:             else if |v.changes| < in :
20:                 v.changes.Add((current, field, u'))
21:             else
22:                 v.copy = Copy(v)
23:                 v.copy.field = u'
24:                 u'.parents[i] = v.copy
25:     return u'
```

Assumimos que a função RawCopy(u) faz e devolve um novo nó com todos os campos com valores idênticos aos de u. As linhas 3-6 então modificam u' para este representar a versão mais atual de u, aplicando todas as modificações de $u.changes$ e atualizando o campo $\mathcal{T}$. O **if** da linha 7 lida com o caso em que u era um nó de entrada da versão $current - 1$, já que neste caso u' passa a ser o nó de entrada correspondente para a versão *current*. O **for** da linha 9 então muda os ponteiros de volta dos nós apontados por u, já que agora u' é o nó ativo que aponta para eles. Por último, o **for** da linha 13 modifica os ponteiros de nós ativos que apontavam para u, fazendo-os apontar para u'. Seja v um nó ativo tal que Access(v, *field*) $= u$, queremos que Access(v, *field*) $= u'$. Isso se reduz a três casos:

1. v foi criado nessa versão — Basta modificar diretamente o campo *field* em v.

2. v tem espaço livre em $v.changes$ — Adicionamos a modificação ao vetor *changes*, ou seja, adicionamos a tripla $(field, current, u')$ a *changes*.

3. v não tem espaço livre em $v.changes$ — Criamos uma cópia de v, usando recursivamente a função COPY, modificamos o campo *field* desta cópia, e atualizamos o vetor *parents* de u' (que ainda aponta para v e não para $v.copy$).

Como cada nó é copiado no máximo uma vez por versão, a função sempre termina.

Análise de espaço e tempo

A ideia do método é copiar os nós quando fazemos modificações em passos de modificação, mas guardam-se algumas modificações no mesmo nó para diminuir o número de cópias necessárias. É claro que, aumentando o tamanho do vetor *changes*, o número de cópias diminui. Vamos mostrar que ter tamanho *in* é o bastante para que o método consuma espaço amortizado $\mathcal{O}(1)$ por passo de modificação.

Utilizaremos o método do potencial. Seja E_i o estado da estrutura (persistente) na i-ésima versão, ou seja, depois da i-ésima operação de modificação. Vamos associar um valor $\Phi(E_i)$ a cada uma das versões da estrutura, com $\Phi(E_0) = 0$ e $\Phi(E_i) \geq 0$ para todo $i > 0$. Seja O_i o número de nós criados pelo método descrito na i-ésima operação de modificação. Então

$$\sum_{i=1}^{m} (O_i + \Phi(E_i) - \Phi(E_{i-1})) = \left(\sum_{i=1}^{m} O_i\right) + \Phi(E_m) - \Phi(E_0) \geq \sum_{i=1}^{m} O_i.$$

Dessa forma, ainda que calcular O_i seja complicado, se escolhemos Φ tal que o cálculo de $O_i + \Phi(E_i) - \Phi(E_{i-1})$ seja simples, conseguimos assim um limite superior para o número de nós criados pelo método, que é $\sum_{i=1}^{m} O_i$.

Queremos escolher Φ de forma que $O_i + \Phi(E_i) - \Phi(E_{i-1})$ seja $\mathcal{O}(M_i)$, onde M_i é o número de passos de modificação durante a i-ésima operação de modificação. Para isso, queremos que o potencial "cancele" o número de nós adicionais criados indiretamente pelas chamadas recursivas de COPY (já que o número de nós criados diretamente em passos de modificação é no máximo M_i). A escolha do potencial é $\Phi(E_i) = A_i \cdot in - L_i$, onde A_i é o número de nós ativos em E_i e L_i é o número de espaços de modificação vazios em todos os nós ativos, ou seja, a soma do valor $in - |u.changes|$ para todo nó ativo u. Note que, como cada nó ativo tem no máximo *in* espaços livres, $\Phi(E_i)$ nunca é negativo, e claramente $\Phi(E_0) = 0$.

Segundo a definição na Seção 7.1, um passo de modificação cria um novo nó, modifica um campo de um nó existente, usando para isso a função de interface MODIFY, ou troca um nó de entrada. Se um passo de modificação cria um novo nó, o potencial não se modifica, já que A_i aumenta em 1 e L_i aumenta em *in*. Além disso, O_i aumenta em 1. Seja N_i o número de tais nós criados durante a i-ésima operação de modificação. Se um nó de entrada é trocado, nem o potencial nem O_i se alteram.

Vamos considerar então nós criados pela função COPY durante a i-ésima operação de modificação. Esta função é uma função é uma função auxiliar e não é chamada diretamente durante os passos de modificação, apenas por MODIFY (linha 4) e recursivamente pela própria COPY (linha 22).

Seja C_i o número de chamadas de COPY feitas por MODIFY e D_i o número de chamadas de COPY feitas por COPY.

Temos que $C_i \leq M_i$, já que MODIFY chama COPY no máximo uma vez, e MODIFY é chamada no máximo uma vez por passo de modificação. Além disso, quando COPY(u) é chamada recursivamente, isso ocorreu pois u tinha o vetor *changes* cheio, logo o potencial diminui em in nesse caso, já que o número de nós ativos (A_i) não muda, mas temos in mais espaços livres (L_i).

O **for** da linha 13 pode adicionar mudanças aos vetores *changes* dos nós que apontam para u (linha 20). Cada adição aumenta Φ em 1. Seja y a quantidade de tais adições. Cada chamada de COPY pode adicionar até in modificações, porém, quando COPY é chamada recursivamente na linha 22, essa adição não é feita, logo $y \leq (C_i + D_i) \cdot in - D_i$. Utilizando a notação definida acima temos:

$$\begin{aligned} O_i + \Phi(E_i) - \Phi(E_{i-1}) &\overset{(1)}{=} N_i + C_i + D_i - D_i \cdot in + y \\ &\overset{(2)}{\leq} N_i + (1 + in) \cdot C_i \\ &\overset{(3)}{\leq} N_i + (1 + in) \cdot M_i \\ &\overset{(4)}{=} \mathcal{O}(M_i) \end{aligned}$$

A igualdade (1) vale pois $O_i = N_i + C_i + D_i$ e pelas mudanças em Φ discutidas, (2) vale pois $y \leq (C_i + D_i) \cdot in - D_i$, e assim cancelamos D_i, (3) vale pois $C_i \leq M_i$ e (4) usa que $N_i \leq M_i$ e in é uma constante.

Isto prova que o espaço total gasto pela estrutura é $\mathcal{O}(\sum_{i=1}^{m} M_i) = \mathcal{O}(M)$, ou seja, espaço amortizado constante por passo de modificação. Além disso, gasta-se tempo constante para criar cada nó, e cada chamada de COPY cria um novo nó. Logo o consumo de tempo dos passos de modificação é limitado pela criação de nós, e os passos de acesso funcionam em tempo $\mathcal{O}(1)$ usando a função ACCESS. Portanto, o tempo total consumido por esse método é $\mathcal{O}(M + A)$, ou seja, tempo amortizado constante por passo de acesso ou modificação.

Concluindo, este método transforma qualquer estrutura ligada com grau de entrada limitado por uma constante em uma estrutura parcialmente persistente, sem aumentar o consumo de tempo assintoticamente, apenas deixando esse tempo amortizado (se já não era). Variações desta técnica também permitem deixar a estrutura totalmente persistente [5], porém, são bem mais complicadas e usam EDs adicionais.

Capítulo 8

Árvore rubro-negra

Apresentaremos neste capítulo a implementação de uma árvore rubro-negra parcialmente persistente utilizando a técnica de node copying, discutida na Seção 7.5. Uma árvore rubro-negra é uma árvore de busca binária balanceada, na qual cada nó é vermelho ou preto, e as cores são usadas para auxiliar no rebalanceamento.

Com uma implementação baseada na de Cormen et al. [3, Cap. 13], a cada inserção ou remoção em uma árvore de n nós, são necessárias apenas $\mathcal{O}(1)$ rotações para rebalancear a árvore, e $\mathcal{O}(\lg n)$ mudanças de cores. Como as cores servem apenas para auxiliar no rebalanceamento, não é necessário que estes dados sejam mantidos em versões que não a atual para persistência parcial. Ou seja, só precisamos saber a cor dos nós ativos, logo são feitos apenas $\mathcal{O}(1)$ passos de modificação (considerando apenas os ponteiros de filho esquerdo e direito de um nó) por inserção, logo esta implementação parcialmente persistente consome tempo $\mathcal{O}((m+a)\lg(m))$ e espaço $\mathcal{O}(m)$, onde m é o número de operações de modificação (inserções e remoções), e a é o número de operações de acesso (acessar um elemento, mínimo, máximo, etc.).

A implementação de Sedgewick e Wayne [13], apesar de ser considerada mais simples, não serve para os nossos propósitos, pois, ao considerar apenas árvores rubro-negras esquerdistas, os algoritmos de inserção e remoção podem realizar $\Theta(\lg n)$ rotações, logo não é possível manter o consumo de espaço $\mathcal{O}(m)$, usando node copying.

Se $\mathcal{O}(m \lg m)$ de espaço é aceitável, uma implementação funcional garante esse espaço, e é totalmente persistente. Esse método é discutido na Seção 7.3, e utilizado nos Capítulos 3, 4, 5 e 6. Basta, em operações de modificação, copiar todo o caminho percorrido na árvore, a partir da raiz.

Estamos interessados na implementação das seguintes operações:

- Insert(*value*)
  Insere um nó com valor *value* na ABB.
- Remove(*value*)
  Remove um nó com valor *value* da ABB.
- Find(*value*)
  Devolve algum nó da ABB com valor *value*. Caso não exista, devolve **null**.

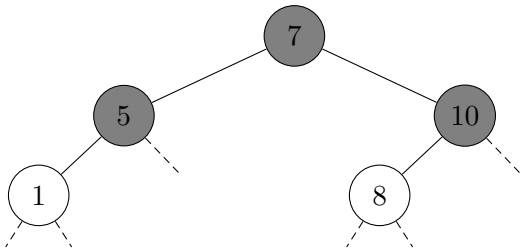

Figura 8.1: Exemplo de árvore rubro-negra. Como no resto das figuras do capítulo, os nós negros têm fundo cinza e os nós vermelhos têm fundo branco. Nas próximas figuras, o valor de cada nó não é especificado.

8.1 Definições

Uma árvore binária consiste de um nó chamado raiz, que tem ponteiro para até duas outras árvores binárias, chamadas subárvore esquerda e subárvore direita. Uma árvore *de busca* binária é uma árvore binária na qual cada nó tem um valor e segue a seguinte propriedade: para cada nó, o valor de todos os nós em sua subárvore esquerda é menor ou igual ao seu valor, e o valor de todos os nós em sua subárvore direita é maior ou igual ao seu valor.

Um nó que não tem algum filho armazena **null** no campo correspondente a esse filho. Uma folha é um nó que não tem filhos, e um link nulo é um campo de filho de algum nó que tem valor **null**. Na Figura 8.1 estes são as arestas tracejadas.

Uma árvore rubro-negra satisfaz as seguintes propriedades rubro-negras:

1. A raiz é negra.
2. Se um nó é vermelho, este não tem filhos vermelhos.
3. Para todo nó, todos os caminhos até links nulos de sua subárvore têm o mesmo número de nós negros.

Considerando apenas os nós negros, a árvore é totalmente balanceada (propriedade 3). Como um nó vermelho não tem filhos vermelhos (propriedade 2), é possível "juntar" os nós vermelhos aos seus pais negros (a raiz não é vermelha pela propriedade 1), e assim cada nó negro fica associado a até três valores (e pode ter até quatro filhos). Logo, uma delimitação superior simples para a altura de uma árvore rubro-negra com n nós é $3\lfloor \lg(n) \rfloor$, pois esta árvore comprimida é totalmente balanceada. Se mantivermos as propriedades rubro-negras a cada inserção ou remoção, e o consumo de tempo destas operações for proporcional à altura da árvore, então o consumo de tempo será $\mathcal{O}(\lg n)$ para todas as operações.

8.2 Implementação da persistência

Em uma árvore rubro-negra efêmera (não persistente), todo nó precisa de dois campos para seus filhos, e um valor booleano que indica se o nó é vermelho ou não. Como já discutido, o campo booleano não precisa ser guardado de forma persistente. Como cada nó tem grau de entrada

no máximo um, na versão persistente é necessário armazenar apenas um ponteiro de volta e um ponteiro extra por nó, dado que usaremos a técnica de node copying.

O ponteiro extra simula o vetor *changes* usado durante a Seção 7.5, que no caso sempre teria zero ou um elemento.

O ponteiro de volta é armazenado no campo *parent*, e é exatamente o ponteiro que aponta para o pai de um nó, o que será conveniente durante a implementação das operações. Note que, de acordo com a implementação da Seção 7.5, o campo *parent* é nulo para nós que não são ativos, logo ele não pode ser utilizado para realizar as operações de acesso, apenas de modificação.

Para evitar a necessidade de replicar código muito parecido, vamos armazenar os filhos como um vetor *child* de duas posições; $u.child[0]$ é o filho esquerdo de u, e $u.child[1]$ é seu filho direito. Dessa forma, casos simétricos podem ser tratados com o mesmo código. Os campos de um nó serão:

- *value* — O valor armazenado no nó.
- *child* — Vetor de filhos.
- *parent* — Ponteiro de volta.
- *red* — Booleano que indica se o nó é vermelho ou não.
- $\mathcal{T}$ — Instante de criação do nó.
- *copy* — Cópia do nó (se ele não é ativo).
- *extra*, *extraSide*, *extra*$\mathcal{T}$ — Ponteiro extra, seu lado e instante de modificação.

É necessário armazenar apenas um ponteiro extra por nó, que pode ser de filho esquerdo ou direito. Note que é necessário armazenar *extraSide*, o lado do ponteiro extra, pois se *extra* for nulo, não há como saber qual filho este é apenas comparando seu valor com o do nó. Para indicar se o ponteiro extra foi utilizado, usaremos que *extra*$\mathcal{T} = -1$ quando não existe tal ponteiro.

8.3 Operações de acesso

Para acessar o campo de um nó na versão *version*, ou na versão atual, usamos versões modificadas das funções Access, dadas no Código 8.1 (neste capítulo, os nomes das funções de interface não estarão sublinhados para não serem confundidos com os nomes das operações da ABB). O campo *current*, como na Seção 7.5, indica o tempo atual da estrutura, ou seja, o número de operações de modificação realizadas, e os campos $\mathcal{T}$ e *extra*$\mathcal{T}$ armazenam o valor de *current* durante a criação do nó e do ponteiro extra, respectivamente.

O Código 8.2 apresenta a operação Find, que devolve um nó com o valor buscado, ou **null** se tal nó não existe. Assumimos que é guardado um vetor *roots*, com o nó de entrada (a raiz) para cada versão.

O algoritmo funciona como em uma árvore efêmera, mas usando a função Child para acessar o filho direito ou esquerdo, em vez de utilizar o campo *child*, que pode não ter a versão desejada do ponteiro. Note que, na linha 4, já utilizamos a notação de Iverson para diminuir o código. Outras operações de acesso (encontrar o maior elemento menor ou igual a algum valor, por exemplo) podem ser implementadas da mesma forma.

Código 8.1: Acesso aos campos de um nó.

Require: u deve ser um nó associado à versão *version*.
1: **function** CHILD(u, *side*, *version*)
2: **if** $u.extra\mathcal{T} \neq -1$ **and** $u.extraSide = side$ **and** $version \geq u.extra\mathcal{T}$:
3: **return** $u.extra$
4: **return** $u.child[side]$
Require: u deve ser ativo ou sua cópia deve ter sido criada na versão atual.
5: **function** CHILD(u, *side*)
6: **return** CHILD(ACTIVE(u), *side*, *current*) ▷ ACTIVE funciona como descrito no Código 7.3.

Código 8.2: Operação FIND em uma ABB rubro-negra parcialmente persistente.

1: **function** FIND(x, *version*)
2: $u = roots[version]$
3: **while** $u \neq$ **null and** $x \neq u.value$:
4: $u =$ CHILD(u, $[x > u.value]$, *version*)
5: **return** u

8.4 Modificação de um campo

O Código 8.3 apresenta as funções MODIFY e COPY, adaptadas da Seção 7.5, para utilizar durante as operações de modificação, que serão apresentadas nas próximas seções. No código, usamos que apenas ponteiros são modificados em nós de uma árvore rubro-negra, nunca o valor dos nós.

Utilizamos que cada nó tem apenas um campo extra e ponteiro de volta para diminuir o código, e o vetor de filhos para evitar duplicação de código. Com estas funções prontas, é possível manter a persistência da estrutura, e nas próximas seções vamos discutir como implementar as operações de inserir e remover um valor de uma árvore rubro-negra.

Note que, na linha 25 do Código 8.3, devemos usar $[\text{CHILD}(v, 1) = u]$ e não $[u'.value > v.value]$ pois o segundo predicado não funciona quando a ABB pode armazenar valores repetidos, já que v pode ter dois filhos com o mesmo valor.

8.5 Inserção em ABB

Para inserir o valor *value* em uma ABB efêmera e não rubro-negra, criamos um novo nó x, com valor *value*. Se a árvore está vazia, fazemos x ser a raiz, e claramente a árvore continua sendo uma ABB válida. Se a árvore não está vazia, seguimos o caminho, a partir da raiz, dado pelo valor *value* (similar a uma operação FIND). Ao analisar um nó, seguimos para seu filho direito se *value* for maior que seu valor, e para seu filho esquerdo caso contrário. Quando encontrarmos um link nulo, substituímos este por x. A árvore continua válida pois era válida inicialmente e, para todo ancestral de x, este está no "lado correto", ou seja, se este ancestral tem valor menor que *value* então x está em sua subárvore direita, e caso contrário em sua subárvore esquerda, pela forma como percorremos a árvore.

Note que é possível existirem valores repetidos na árvore, por isso na definição de ABB usamos que os nós na subárvore esquerda têm valores menores ou iguais e os nós da subárvore direita têm

Código 8.3: Funções MODIFY e COPY, adaptadas da Seção 7.5.

Require: v deve ser **null** ou não ter pai.

```
1:  function MODIFY(u, side, v)                    ▷ Faz v filho de u do lado side.
2:      u = ACTIVE(u)
3:      if u.T < current :
4:          u.copy = COPY(u)
5:          u = u.copy
6:      if u.child[side] ≠ null :
7:          u.child[side].parent = null
8:      u.child[side] = ACTIVE(v)
9:      if u.child[side] ≠ null :
10:         u.child[side].parent = u
11: function COPY(u)
12:     u' = RAWCOPY(u)
13:     u'.T = current
14:     if u.extraT ≠ −1 :
15:         u'.child[u.extraSide] = u.extra
16:         u'.extraT = −1                         ▷ Limpando o campo extra.
17:     if roots[current] = u :
18:         roots[current] = u'
19:     for side ∈ {0, 1} :
20:         if u'.child[side] ≠ null :
21:             u'.child[side].parent = u'
22:     u.parent = null
23:     if u'.parent ≠ null :
24:         v = u'.parent
25:         side = [CHILD(v, 1) = u]
26:         if v.T = current :
27:             v.child[side] = u'
28:         else if v.extraT = −1 :
29:             v.extraT = current
30:             v.extra = u'
31:             v.extraSide = side
32:         else
33:             v.copy = COPY(v)
34:             v.copy.child[side] = u'
35:             u'.parent = v.copy
36:     return u'
```

valores maiores ou iguais ao valor do nó. O Código 8.4 mostra a implementação da operação INSERT.

Código 8.4: Inserção em uma ABB efêmera e não rubro-negra.

```
1:  function INSERT(value)
2:      x = new NODE(value)                    ▷ Nó com valor value e outros campos vazios.
3:      if root = null :
4:          root = x
5:      else
6:          u = root
7:          while u ≠ null :                   ▷ v é o pai de u.
8:              v = u
9:              u = u.child[[value > u.value]]
10:         v.child[[value > v.value]] = x
```

8.6 Inserção em rubro-negra

A inserção em uma árvore rubro-negra começa exatamente como a de uma ABB simples, porém pintamos x de vermelho. Se a árvore era vazia, x é a raiz e apenas a propriedade 1 é violada. Caso contrário, apenas a propriedade 2 pode estar sendo violada, caso o pai de x seja vermelho. A propriedade 3 sempre continua a valer, pois adicionamos um nó vermelho.

Para arrumar a possível violação da propriedade 1 ou 2, vamos fazer um laço com as seguintes invariantes (valem no começo da iteração do laço):

(A) x é vermelho.

(B) Se x é a raiz, a propriedade 1 é violada.

(C) Se x tem pai vermelho, a propriedade 2 é violada.

(D) As propriedades não são violadas por outros nós.

Numa iteração deste laço, vamos ou acabar com todas as violações, ou mudar x para um nó de altura menor. Dessa forma "subimos" a violação, e conseguimos acabar com todas as violações em tempo $\mathcal{O}(\lg n)$, pois a árvore é balanceada.

Se, no começo da iteração, a raiz é x, então para arrumar esta violação basta pintar x de preto. Caso contrário, vamos recolorir ou rotacionar nós, sempre mantendo a propriedade 3, para diminuir a altura do nó que viola as propriedades.

Rotações

Rotações são modificações locais de nós de uma ABB que mantém as propriedades de uma ABB. Elas serão usadas para "subir" a violação na árvore rubro-negra. Uma rotação troca um nó u por um de seus filhos, fazendo as modificações necessárias para manter as propriedades de uma ABB.

A Figura 8.2 mostra uma rotação genérica. Note que, nos dois lados da figura, com certo abuso de notação, temos $\alpha \leq u \leq \beta \leq v \leq \gamma$, isto é, se as propriedades de ABBs são seguidas em um lado da figura, elas continuam a ser respeitadas no outro lado da figura, após a rotação. Dizemos que u é rotacionado em direção a v no caso da chamada ROTATE$(u, 1)$.

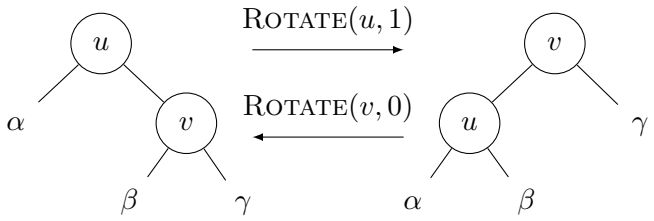

Figura 8.2: Exemplo de rotação direita e esquerda em uma ABB. Os símbolos α, β e γ indicam subárvores, possivelmente vazias.

Subindo a violação

Se a árvore ainda tem violações no começo da iteração do laço, se x não é a raiz, seja y o pai de x. Temos que y é vermelho, logo não é a raiz (pois a propriedade 1 não é quebrada), e por isso também tem um pai z (o avô de x). Como x é o único nó que viola alguma propriedade, seu avô z com certeza é preto. Seja w o tio de x (o filho de z que não é y).

Caso 1. w existe e é vermelho. Podemos apenas trocar as cores de y, w e z para arrumar a violação de x (Figura 8.3). Todos os caminhos até links nulos que passam por z têm que passar por y ou w, e como agora z é vermelho e y e w são negros, o número de nós negros nestes caminhos continuam os mesmos, e então a propriedade 3 se mantém.

O nó x não viola mais nenhuma propriedade, mas o nó z se tornou vermelho e pode ter o pai vermelho ou ser a raiz, logo a próxima iteração deve ter $x' = z$.

Caso 2. w não existe ou é preto, e x e y são filhos de mesmo lado (por exemplo, y é filho esquerdo de z e x é filho esquerdo de y). Então realizamos uma rotação de z em direção a y, e trocamos as cores de y e z (Figura 8.4).

Antes da transformação: os caminhos até links nulos que passam por x ou α passam apenas pelo nó negro z nesta parte do caminho, e os caminhos que passam por w passam pelos nós pretos z e w (se existir). Após a transformação: os caminhos até links nulos que passam por x ou α passam apenas pelo nó negro y, e os caminhos que passam por w passam pelos nós pretos y e w (se existir). Logo a propriedade 3 é mantida. O nó y é preto, logo não é possível que este viole as propriedades 1 ou 2, e por isso não existem mais iterações depois desta.

Caso 3. w não existe ou é preto, e x e y são filhos de lados diferentes. A rotação feita no caso 2 não permite manter a propriedade 3, então primeiro fazemos uma rotação em y na direção de x (Figura 8.5), e assim transformamos este caso no caso 2 (se trocamos x por y).

Se o caso 2 ou 3 é executado, não existe mais violações de propriedade e o laço acaba. Se o caso 1 é executado, trocamos x por um nó de altura menor (que pode ser a raiz). Logo, o caso 1 ocorre no máximo h vezes, onde h é a altura da árvore. Como cada caso consiste apenas de rotações e mudanças de cores, e rotações podem ser feitas em tempo constante (são apenas algumas mudanças de ponteiros), a inserção em uma árvore rubro-negra efêmera de n nós consome tempo $\mathcal{O}(\lg n)$, já que árvore é balanceada.

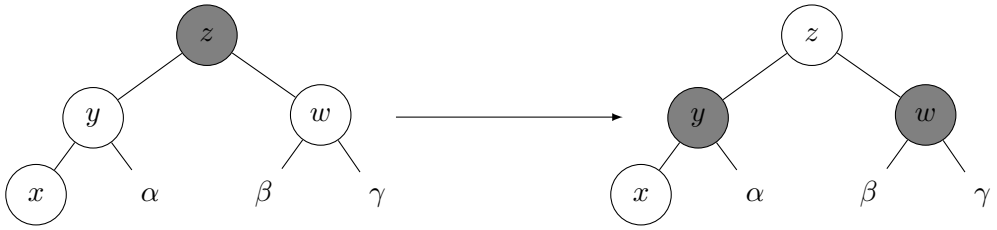

Figura 8.3: Aplicação do caso 1. Não importa se x é filho direito ou esquerdo.

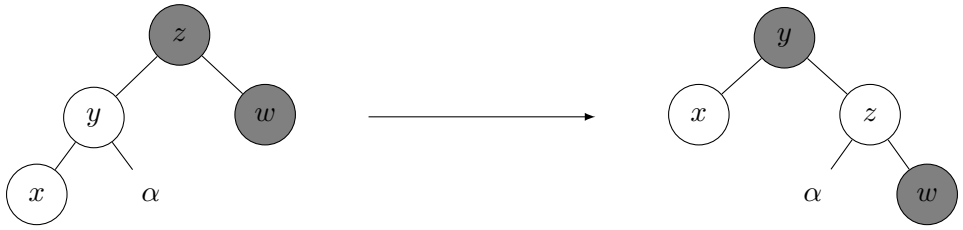

Figura 8.4: Aplicação do caso 2, assumindo que x é filho esquerdo. Os nós x e w (se existir) podem ter filhos.

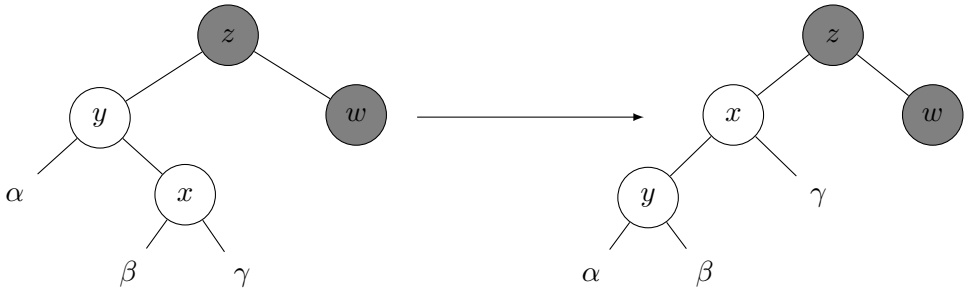

Figura 8.5: Aplicação do caso 3, assumindo que x é filho direito. O nó w, se existir, pode ter filhos.

Implementação e persistência parcial

A maior parte das modificações envolvidas em uma inserção são mudanças de cores, que não precisam ser guardadas de forma persistente. Mudança de ponteiros só ocorrem para adicionar o nó x como filho de outro nó, e em possíveis rotações para tratar os casos 2 e 3. Como discutido, após um caso 2 ou 3, o algoritmo acaba, logo são feitas apenas $\mathcal{O}(1)$ mudanças de ponteiros, e utilizando a persistência por node copying conseguimos consumo de espaço amortizado constante por inserção.

Código 8.5: Inserção em árvore rubro-negra parcialmente persistente.

```
1:  function INSERT(value)
2:      current = current + 1
3:      roots[current] = roots[current − 1]
4:      x = new NODE(value)            ▷ Nó vermelho com valor value e outros campos vazios.
5:      x.T = current
6:      if roots[current] = null :
7:          roots[current] = x
8:      else
9:          u = roots[current]
10:         while u ≠ null :                                          ▷ v é o pai de u.
11:             v = u
12:             u = CHILD(u, [value > u.value])
13:         MODIFY(v, [value > v.value], x)
        ▷ Arrumando a possível violação causada pelo nó x.
14:     while x.red and x.parent ≠ null and x.parent.red :
15:         y = x.parent
16:         z = y.parent
17:         sideX = [CHILD(y, 1) = x]
18:         sideY = [CHILD(z, 1) = y]
19:         w = CHILD(z, not sideY)
20:         if w ≠ null and w.red :                                   ▷ Caso 1.
21:             z.red = true
22:             y.red = false
23:             w.red = false
24:             x = z
25:         else
26:             if sideX ≠ sideY :                                    ▷ Caso 3.
27:                 ROTATE(y, sideX)
28:                 x, y = y, x                                       ▷ Trocando x e y.
29:             ROTATE(z, sideY)                                      ▷ Caso 2.
30:             ACTIVE(z).red = true
31:             ACTIVE(y).red = false
32:             break
33:     roots[current].red = false        ▷ Caso a raiz tenha sido pintada de vermelha.
```

O Código 8.5 mostra então a implementação da inserção, como discutida, dado que a função ROTATE funciona corretamente. As linhas 4-13 inserem o nó na árvore como em uma ABB normal, mas usando as funções CHILD e MODIFY para manter a persistência. As linhas 14-33 fazem o laço discutido nesta seção.

No começo da primeira iteração do laço, as propriedades valem pois x é um nó vermelho que

recém-inserido. As linhas 15-18 determinam os nós y, z e w, usados nos casos. O caso 1 é tratado nas linhas 20-24, trocando as cores de z, y e w e fazendo o x da próxima iteração ser z, pois é o único nó que possivelmente viola alguma propriedade.

O caso 3 é transformado no caso 2 nas linhas 26-28, fazendo uma rotação de y na direção de x, e trocando esses 2 nós. As linhas 29-32 então tratam o caso 2, fazendo uma rotação de z em direção à y e trocando a cor destes, como na Figura 8.4. As chamadas a ACTIVE nas linhas 30 e 31 tratam o caso quando as rotações causaram cópias, e por isso y ou z não são mais ativos, pois para modificar o campo efêmero *red* devemos modificar o nó ativo. Nestes dois casos o **break** termina o laço.

Por último, a linha 33 pinta a raiz de preto, já que o laço não trata esse caso e termina quando u não tem pai.

A função consome tempo amortizado $\mathcal{O}(\lg n)$ pois a altura da árvore é $\mathcal{O}(\lg n)$, ambos os lações das linhas 10 e 14 realizam um número de iterações proporcional a altura, e a função ROTATE e MODIFY consomem tempo (e espaço) amortizado constante. Como essas funções só são chamadas um número constante de vezes, o consumo de espaço é amortizado $\mathcal{O}(1)$.

O Código 8.6 mostra a implementação da função ROTATE, que usa apenas um número constante de chamadas a MODIFY. Note que é necessário, entre quaisquer duas chamadas, manter que cada nó tem no máximo um outro nó que aponta para ele, logo é preciso ter mais cuidado que com a implementação em uma ABB efêmera. Apesar das funções CHILD e MODIFY funcionarem se receberem nós que já foram copiados nesta operação, na função ROTATE é necessário acessar o campo *parent* de u, por isso na linha 6 trocamos u por sua cópia, se existir.

Código 8.6: Rotação em uma árvore rubro-negra parcialmente persistente.

Require: u deve ter um filho *side* na versão *current*.

```
1:  function ROTATE(u, side)
2:      v = CHILD(u, side)
3:      β = CHILD(v, not side)
4:      MODIFY(v, not side, null)
5:      MODIFY(u, side, β)
6:      u = ACTIVE(u)
7:      if u.parent ≠ null :
8:          MODIFY(u.parent, [CHILD(u.parent, 1) = u], v)
9:      else
10:         roots[current] = ACTIVE(v)          ▷ Se u não tiver pai, u é a raiz.
11:     MODIFY(v, not side, u)
```

8.7 Remoção em ABB

Remover um nó é mais complicado que inserir, pois o nó que desejamos excluir pode ter filhos, então não basta removê-lo, temos que substituí-lo por algum de seus descendentes, e ainda manter as propriedades de uma ABB. Para remover o nó u, temos dois casos.

Se u não tem filho direito, então podemos apenas substituí-lo por seu filho esquerdo (que pode ser **null**), pois assim as propriedades de ABB continuam sendo seguidas.

Se u tem filho direito, seja x o nó de menor valor na subárvore direita de u (ou seja, o nó com menor valor maior que o valor de u). O nó x não tem filho esquerdo (ou existiria um nó com

valor menor que o dele na mesma subárvore), logo podemos substituir x por seu filho direito, e substituir u por x.

As propriedades de ABB continuam a ser seguidas pois todos os elementos da subárvore direita de u têm valores maiores ou iguais aos de x (pois este era o mínimo desta subárvore), e os elementos da subárvore esquerda de u tem valores menores ou iguais aos de x pois este era da subárvore direita de u. No Código 8.7, assumimos que os nós têm ponteiro de pai em seu campo *parent*.

Código 8.7: Remoção em uma ABB efêmera e não rubro-negra.

```
1:  function MinElement(u)
2:      while u.child[0] ≠ null :
3:          u = u.child[0]
4:      return u
5:  function Transplant(u, x)
6:      v = u.parent
7:      if v ≠ null :
8:          v.child[[v.child[1] = u]] = x
9:      else
10:         root = x
11:     if x ≠ null :
12:         x.parent = v
Require: A árvore tem um nó com valor value.
13: function Remove(value)
14:     u = Find(value)
15:     if u.child[1] = null :
16:         Transplant(u, u.child[0])
17:     else
18:         x = MinElement(u.child[1])
19:         Transplant(x, x.child[1])
20:         Transplant(u, x)
21:         x.child = u.child
```

A função MinElement(u) devolve um nó com menor valor da subárvore de u. Pelas propriedades de uma ABB, este está na subárvore esquerda de u se esta existe, então o **while** segue links de esquerda até encontrar o menor elemento.

A função Transplant(u, x) substitui o nó u pelo nó x. Para fazer isso, ela faz o pai de u apontar para x em vez de u (tratando o caso quando u é raiz). A função não modifica os filhos de u ou x.

A linha 14 busca pelo nó u com valor *value*. O **if** da linha 15 trata o caso em que u não tem filho direito, substituindo-o por seu filho esquerdo, usando a função Transplant. Note que, ao final da função, o nó u aponta para um filho que não tem ponteiro de pai para u, mas isto não é um problema pois o nó u foi removido da árvore.

Se u tem filho direito, a linha 18 busca o nó x, mínimo da subárvore direita de u. Este nó é trocado por seu filho direito e u é trocado por este nó. Após a chamada de Transplant(u, x), o nó x é inválido pois, apesar de estar no lugar de u, tem ponteiros em *child* que não são válidos, por isso a linha 21 copia os filhos de u para x. Note que o código funciona mesmo quando x é o próprio filho direito de u.

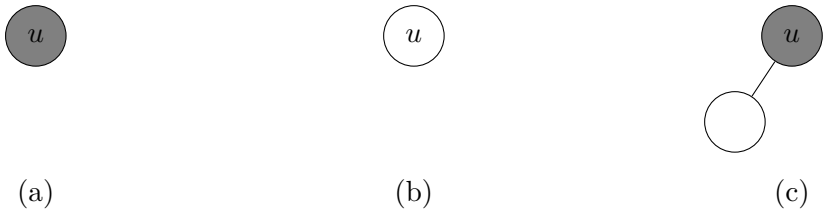

Figura 8.6: Possíveis subárvores rubro-negras com raiz u que não tem filho direito.

8.8 Remoção em rubro-negra

Para remover um nó de uma árvore rubro-negra, fazemos como em uma ABB normal, mas é necessário se preocupar com as propriedades rubro-negras. Quando u tem filho direito, ao substituir u por x, podemos copiar a cor de u para x, e assim essa operação não viola nenhuma propriedade local a u. Porém, a substituição de x por seu filho direito ou, no caso em que u não tem filho direito, a substituição de u por seu filho esquerdo, podem causar violações das propriedades em torno de x e u. Nesses casos, porém, como x ou u não tem um dos filhos, a estrutura da subárvore é bem simples. Trataremos o caso de u não ter filho direito, o outro caso é simétrico.

Pelas propriedades 2 e 3, quando um nó u não tem o filho direito, a estrutura de sua subárvore pode ser apenas uma das três ilustradas na Figura 8.6. Se a versão Transplant(u, x) de árvores rubro-negras também copiar a cor de u para x, caso este não seja nulo, todas as propriedades rubro-negras continuam a ser respeitadas nas árvores (b) e (c), quando u é substituído por seu filho esquerdo. Na árvore (a), u é removido e substituído por **null**, logo os caminhos até links nulos que passavam por u agora têm um nó negro a menos, exceto se u for a raiz. Nesse caso, a árvore final é vazia e segue todas as propriedades. Usamos então a função auxiliar AddBlack, que arruma a violação da propriedade 3, e será discutida na próxima seção.

O Código 8.8 mostra a implementação da remoção, e funciona de forma similar a de uma ABB qualquer. A operação Find é usada para encontrar um nó com valor *value*. A função MinElement funciona como anteriormente, mas usando Child para acessar os filhos. Já Transplant(u, x) foi modificada para remover o ponteiro que aponta para x, caso esse exista. Isto faz diferença no caso da ABB (parcialmente) persistente pois queremos manter a propriedade de que no máximo um nó aponta para x a cada passo. A função Transplant também copia a cor de u para x, caso este não seja nulo.

Se u não tem filho direito, no **if** da linha 22 este é trocado pelo seu filho esquerdo e, como discutido, se u é preto e não tem filho esquerdo, ocorre uma violação da propriedade 3 (a menos que u seja a raiz), que é então consertada por AddBlack. Esta função recebe um nó e qual a direção do filho que deveria ser preto; isso é feito pois esse filho pode na verdade ser um link nulo. A função então modifica a árvore, rotacionando e mudando cores, de forma a remover a violação da propriedade 3. Discutiremos a implementação desta função nas próximas subseções.

Se u tem filho direito, o processo funciona assim como em uma ABB qualquer, mas chamamos AddBlack caso a substituição de x tenha causado uma violação. O nó y é o pai de x, exceto quando x é o próprio filho direito de u, pois neste caso $x.parent = u$ no começo do bloco, mas o nó u será substituído pelo nó x, logo fazemos $y = x$, pois este será o nó que aponta para o link nulo que deveria ser preto (se a função AddBlack for chamada).

Código 8.8: REMOVE em árvore rubro-negra parcialmente persistente.

```
1: function MinElement(u)
2:     while Child(u, 0) ≠ null :
3:         u = Child(u, 0)
4:     return u
5: function Transplant(u, x)
6:     x = Active(x)
7:     if x ≠ null and x.parent ≠ null :                ▷ Removendo link para x, se houver.
8:         Modify(x.parent, [Child(x.parent, 1) = x], null)
9:     u = Active(u)
10:    v = u.parent
11:    if v ≠ null :
12:        Modify(v, [Child(v, 1) = u], x)
13:    else
14:        roots[current] = x
15:    if x ≠ null :
16:        x.red = u.red
Require: A árvore tem um nó com valor value.
17: function Remove(value)
18:    current = current + 1
19:    roots[current] = roots[current − 1]
20:    u = Find(value, current)
21:    v = u.parent
22:    if Child(u, 1) = null :
23:        needFix = (v ≠ null and not u.red and Child(u, 0) = null)
24:        Transplant(u, Child(u, 0))
25:        if needFix :
26:            AddBlack(v, [Child(v, 1) = null])
27:    else
28:        x = MinElement(Child(u, 1))
29:        if x = Child(u, 1) :
30:            y = x
31:        else
32:            y = x.parent
33:        needFix = (not x.red and Child(x, 1) = null)
34:        Transplant(x, Child(x, 1))
35:        Transplant(u, x)
36:        for side ∈ {0, 1} :
37:            child = Child(u, side)
38:            Modify(u, side, null)
39:            Modify(x, side, child)
40:        if needFix :
41:            AddBlack(y, [Child(y, 1) = null])
```

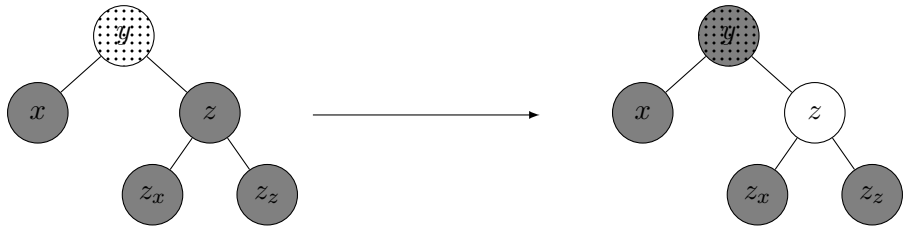

Figura 8.7: Aplicação do caso 1, não importa se x é filho direito ou esquerdo.

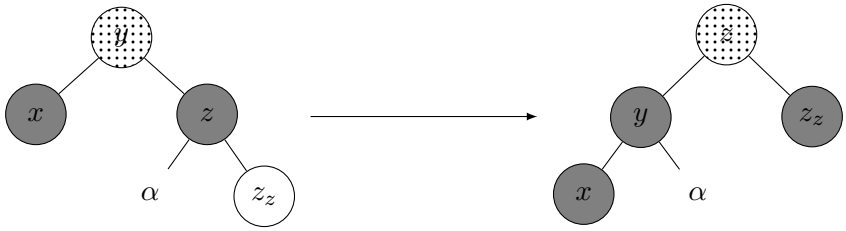

Figura 8.8: Aplicação do caso 2, assumindo que x é filho esquerdo.

Subindo violações

Na função AddBlack(y, *side*), assim como no final da inserção, teremos um laço que, a cada iteração, ou termina com todas as violações, ou de certa forma "sobe" essas violações. A violação é da propriedade 3, pois todos os caminhos de y a links nulos seguindo seu filho *side* têm um nó negro a menos que os caminhos passando por seu outro filho.

Seja x o filho de y na direção *side* (pode ser nulo), e z seu filho na outra direção. Então se x existe e é vermelho, basta pintá-lo de preto e a propriedade 3 voltará a ser satisfeita. Caso contrário consideraremos alguns casos. Note que o filho z sempre existe, pois os caminhos começando em y e indo na direção de z precisam ter pelo menos um nó negro. Sejam z_x e z_z os filhos de z do lado *side* e do outro lado, respectivamente.

Caso 1. z é preto, z_x e z_z ou não existem ou são negros.

A Figura 8.7 mostra esse caso. O nó pontilhado indica que este nó pode ser tanto vermelho quanto negro. Todas as figuras assumem que x é filho esquerdo. O nó x pode não existir, este representa apenas uma direção a partir de y.

Pintamos z de vermelho e y de preto. Dessa forma, como os filhos de z são negros (ou não existem), a propriedade 2 não é quebrada, e temos um nó negro a mais nos caminhos

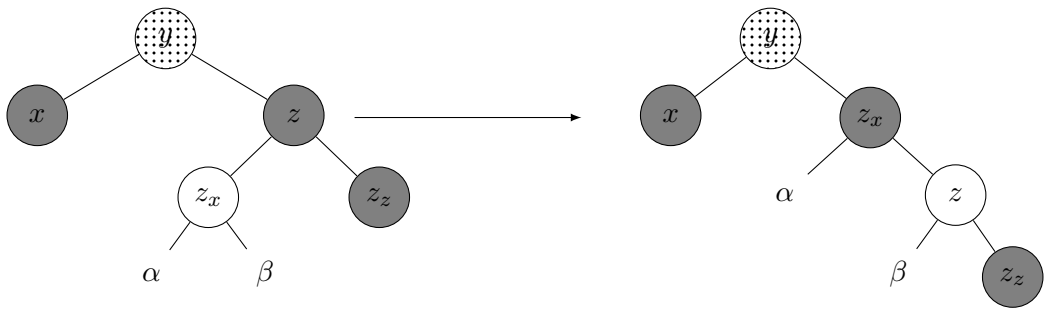

Figura 8.9: Aplicação do caso 3, assumindo que x é filho esquerdo.

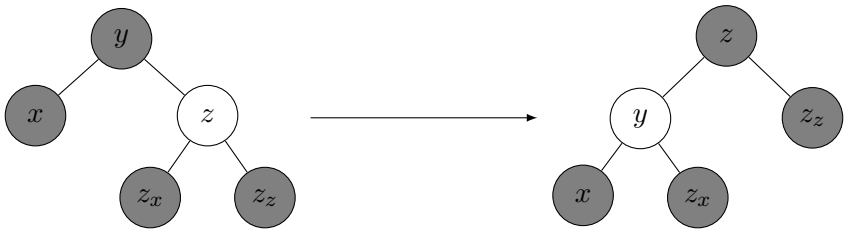

Figura 8.10: Aplicação do caso 4, assumindo que x é filho esquerdo.

que passam por y e x. Se y é vermelho, o laço termina pois todas as propriedades foram restauradas, caso contrário, não é possível pintar y de preto duas vezes, e os caminhos que vão do pai de y para y têm um preto a menos. Assim, recomeçamos o laço trocando y por seu pai e *side* pelo lado apropriado.

Se y for a raiz, o pai desta é **null**, e o laço deve acabar neste caso, já que não é um problema todos os caminhos a partir da raiz terem um preto a menos. Isso apenas significa que a altura negra (quantidade de nós negros dos caminhos da raiz até qualquer link nulo) da árvore diminui após esta remoção.

Caso 2. z é preto, z_z é vermelho.

Rotacionamos y em direção a z, trocamos a cor destes dois nós e pintamos z_z de preto. A Figura 8.8 ilustra esse caso. Note que os caminhos nesta subárvore que passam por α e z_z contém o mesmo número de nós negros (independentemente da cor de y), e os caminhos que passam por x têm um nó negro a mais. Logo, a propriedade 3 volta a valer. Além disso, a propriedade 2 continua valendo pois os nós modificados são negros, exceto talvez por z, mas este tem a mesma posição e cor que y, e a propriedade 2 não era violada no início da iteração, logo continua não violada.

Caso 3. z é preto, z_x é vermelho e z_z é preto ou não existe.

Rotacionamos z em direção a z_x, e trocamos a cor destes dois nós. A Figura 8.9 ilustra esse caso. Note que os caminhos que passam por α, β e z_z continuam com o mesmo número de nós negros, logo a propriedade 3 não é violada nesses nós (mas continua sendo violada por x). Após estas modificações, o caso 2 pode ser aplicado, já que x tem um irmão negro (este nó é z_z) com filho do lado contrário a x vermelho (este nó é z).

Caso 4. z é vermelho.

Como os caminhos até links nulos de y na direção de z têm pelo menos um nó negro, z_x e z_y existem e são negros (já que a propriedade 2 não é violada). Além disso, y é preto (já que z é vermelho). Assim, rotacionamos y na direção de z e trocamos a cor destes dois nós. Note que os caminhos que passam por x, z_x e z_z continuam com o mesmo número de nós negros.

A Figura 8.10 ilustra esse caso. O novo irmão de x é preto (este nó é z_x), e por isso algum dos outros casos (1, 2, ou 3) se aplica. Note que, se o caso 1 se aplica, como y é vermelho, este pode ser pintado de preto, e por isso o laço termina após a aplicação deste caso.

Não é imediatamente claro que os quatro casos cobrem todas as possibilidades. Como discutido, existe pelo menos um nó negro em todos os caminhos que passam por y em direção a z, logo z

existe. Se z é vermelho, estamos no caso 4. Caso contrário, z é preto. Se os filhos de z são ambos negros ou não existem (note que, pela propriedade 3, não é possível que um não exista e o outro seja preto), estamos no caso 1. Caso contrário, algum dos filhos de z é vermelho (ou ambos). Se z_z é vermelho, estamos no caso 2, e caso contrário z_x é vermelho e z_z não, logo estamos no caso 4.

Portanto, sempre um dos casos é aplicável. Se o caso 1 é aplicado, algumas mudanças de cores são feitas; se y era vermelho o laço termina, e caso contrário o laço continua, mas a altura de y diminui. Se o caso 2 é aplicado, as violações são removidas e o laço termina. Se o caso 3 é aplicado, segue uma imediata aplicação do caso 2 e o laço também termina. Se o caso 4 é aplicado, segue uma aplicação de algum dos outros casos. Se esta aplicação for do caso 2 ou 3, o laço termina depois de algumas rotações. Se for uma aplicação do caso 1, como y é vermelho após a aplicação do caso 4, o laço vai terminar após este caso.

Dessa forma, exceto por mudanças de cor, apenas um número constante de mudanças de campos são feitas durante uma remoção, para substituir o nó no começo da remoção e para terminar o laço durante a chamada de AddBlack. Portanto, usando Child e Modify para manter a persistência (parcial), a operação de remoção consome tempo $\mathcal{O}(\lg n)$ e espaço $\mathcal{O}(1)$.

Código 8.9: Implementação de AddBlack.

```
1:  function AddBlack(y, side)
2:      y = Active(y)                                   ▷ Versão mais atual de y.
3:      while y ≠ null :
4:          z = Child(y, not side)
5:          if z.red :                                  ▷ Arrumando caso 4 para caso 1, 2 ou 3.
6:              Swap(y.red, z.red)                      ▷ Trocando cores.
7:              Rotate(y, not side)
8:              y = Active(y)
9:              z = Child(y, not side)
10:         z_x = Child(z, side)
11:         z_z = Child(z, not side)
12:         if (z_x = null or not z_x.red) and (z_z = null or not z_z.red) :   ▷ Caso 1.
13:             z.red = true
14:             if y = roots[current] or y.red :
15:                 y.red = false
16:                 break
17:             else
18:                 side = [Child(y.parent, 1) = y]
19:                 y = y.parent
20:         else
21:             if z_x ≠ null and z_x.red :             ▷ Arrumando caso 3 para caso 2.
22:                 Swap(z.red, z_x.red)
23:                 Rotate(z, side)
24:                 y = Active(y)
25:                 z = Child(y, not side)
26:                 z_z = Child(z, not side)
27:             Swap(y.red, z.red)                      ▷ Caso 2.
28:             z_z.red = false
29:             Rotate(y, not side)
30:             break
```

	Node copying	Funcional
INSERT(*value*)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
REMOVE(*value*)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
FIND(*value*)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 8.1: Comparação do consumo de tempo e espaço da solução implementada neste capítulo e de uma implementação funcional, feita como indicado na Seção 7.3, onde n é o tamanho da ABB. Note que a implementação deste capítulo é parcialmente persistente, enquanto a implementação funcional é totalmente persistente.

A implementação da função ADDBLACK no Código 8.9 segue os casos descritos nesta seção. É necessário cuidado ao modificar a cor de um nó, já que é necessário fazer essa modificação na versão mais atual do nó. Por isso, na aplicação dos casos, as mudanças de cores são feitas antes das rotações (que podem gerar cópias dos nós), e após as rotações as versões mais novas de cada nó são atualizadas.

A primeira coluna da Tabela 8.1 mostra o consumo de tempo e espaço da implementação discutida neste capítulo.

Capítulo 9

Localização de ponto

Neste capítulo analisaremos uma variação do problema de localização de ponto (point location) [12], e mostraremos uma solução utilizando a ABB persistente descrita no Capítulo 8.

Dado um conjunto de polígonos $\{P_1, \ldots, P_k\}$ tal que nenhum dos polígonos se intersecta, queremos responder múltiplas consultas do seguinte tipo: Dado um ponto p, determine i tal que $p \in P_i$ ou diga que tal i não existe.

A Figura 9.1 mostra um exemplo do problema com três polígonos. Os pontos de consulta estão coloridos com a cor do polígono a qual pertencem, ou pretos se não pertencem a nenhum dos polígonos.

9.1 Solução ingênua

Note que, neste problema, temos os polígonos de antemão, e queremos pré-processá-los de forma a poder responder as consultas de maneira rápida. Considere $n \coloneqq \sum_{i=1}^{k} |P_i|$, ou seja, n é o número total de vértices em todos os polígonos. A solução mais simples para o problema é, para cada consulta, verificar para cada polígono se o ponto está no polígono. Esta solução tem complexidade $\langle \mathcal{O}(1), \mathcal{O}(n) \rangle$, usando a mesma notação de tempo de pré-processamento e consulta da Seção 1.2.

Nessa solução, para determinar em qual polígono está um ponto, determinamos qual segmento

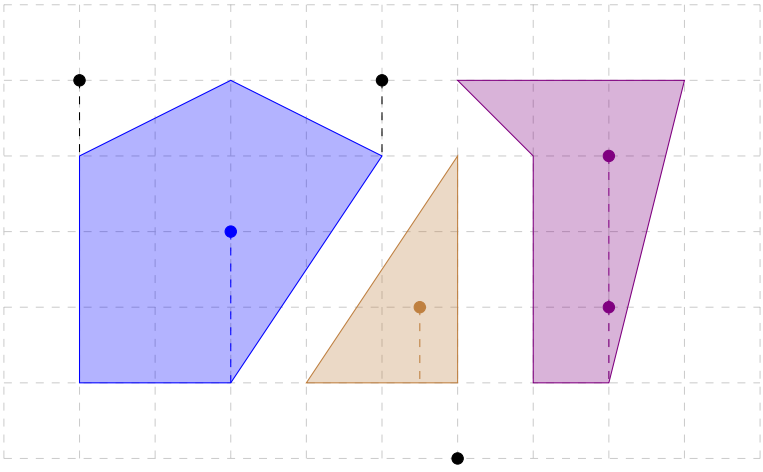

Figura 9.1: Exemplo do problema de localização de ponto.

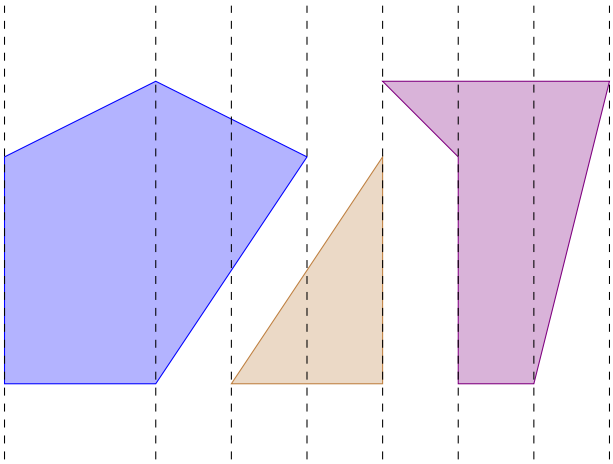

Figura 9.2: Partição do exemplo em faixas.

dos polígonos está diretamente abaixo do ponto, e analisando esse segmento determinamos se o ponto está dentro do polígono que contém aquele segmento ou fora de todos os polígonos. Na Figura 9.1, a projeção de cada ponto no segmento diretamente abaixo está indicada pelas linhas tracejadas. Um ponto que não tem nenhum segmento abaixo dele, como o ponto mais abaixo no exemplo, não pertence a nenhum polígono.

9.2 Partição do plano em faixas

A solução de Cole [2] envolve particionar o plano por retas verticais passando pelos vértices dos polígonos, como na Figura 9.2. Em cada uma das faixas da partição, os segmentos presentes naquela faixa são ordenados verticalmente, ou seja, se sabemos em qual faixa está o ponto da consulta (o que pode ser determinado por uma busca binária), é possível determinar o segmento diretamente abaixo deste ponto usando busca binária nos segmentos presentes nesta faixa. Se armazenarmos os segmentos de cada faixa de forma ingênua, isto se torna uma solução com complexidade $\langle \mathcal{O}(n^2 \lg n), \mathcal{O}(\lg n) \rangle$, como na solução de Dobkin e Lipton [4].

A observação essencial para reduzir a complexidade da solução é que a diferença entre duas faixas adjacentes consiste apenas dos segmentos que terminam ou começam entre estas duas faixas. Além disso, se considerarmos as faixas da esquerda para a direita, cada segmento é adicionado e removido apenas uma vez. Podemos então usar uma linha de varredura para "visitar" todas as listas de cada faixa usando apenas n adições e remoções.

Queremos manter os segmentos ordenados, e adicionar e remover segmentos ao longo do tempo; para isso é possível utilizar uma ABB que armazena os segmentos, e é possível nesta ABB realizar uma busca para determinar o segmento diretamente abaixo de algum ponto, quando estamos na faixa correspondente a esse ponto. Note que não são quaisquer dois segmentos que são comparáveis (apenas se eles são intersectados por uma mesma reta vertical), porém, a todo momento, todos os segmentos presentes na ABB são comparáveis entre si, e essa condição é suficiente para a utilização de uma ABB.

9.3 Conversão de offline em online

Se tivéssemos os pontos das consultas de antemão, poderíamos separá-los por faixas e, ao percorrer os segmentos da esquerda para a direita, usar a ABB para determinar a resposta para cada uma das consultas.

Podemos, entretanto, usar uma ABB parcialmente persistente, percorrer os segmentos da esquerda para a direita de forma que, quando dada uma consulta para um ponto p, podemos acessar a versão da ABB que corresponde à faixa que contém p. Dessa forma, transformamos uma solução offline, que necessitava ter os pontos de antemão, em uma solução online, que pode responder consultas imediatamente.

Usando a estrutura apresentada no Capítulo 8, a solução para o problema tem complexidade $\langle \mathcal{O}(n \lg n), \mathcal{O}(\lg n) \rangle$, e usa espaço $\mathcal{O}(n)$. Os detalhes da implementação serão discutidos nas próximas seções.

9.4 Pré-processamento

Essa solução tem alguns casos de borda, por exemplo, quando o ponto de consulta está exatamente na borda de duas faixas, ou quando existem segmentos verticais. Para evitar estes problemas, consideramos que um ponto p está à esquerda de q se tem coordenada x menor, ou se a coordenada x é igual e a coordenada y é menor. Assumimos aqui que os pontos de consulta não podem ser iguais a pontos dos polígonos, e esse caso pode ser resolvido separadamente de forma simples.

São dados polígonos $\{P_1, \ldots, P_k\}$. No pseudocódigo, usamos que $|P_i|$ é o número de pontos do i-ésimo polígono e P_i^j é o j-ésimo ponto do polígono P_i. Para facilitar o código vale que $P_i^{|P_i|+1} = P_i^1$ e $P_i^0 = P_i^{|P_i|}$. Assumimos que os pontos dos polígonos são dados em sentido anti-horário.

Um segmento é um objeto com quatro campos: *from* e *to*, seus pontos de início e fim, *polygon*, a qual polígono pertence este segmento, e *top*, um booleano que indica se o segmento é da parte "de cima" do polígono, ou seja, se os pontos imediatamente acima desse segmento não pertencem a nenhum polígono. Assumimos que *from* é sempre o "ponto esquerdo" do segmento. Usamos SEGMENT(*polygon*, *from*, *to*, *top*) para inicializar os campos de um segmento.

Precisamos percorrer as arestas dos polígonos da esquerda para a direita. Para isso, ordenamos todos os pontos de todos os polígonos usando o vetor *points* e, ao processar um ponto, adicionamos ou removemos cada um dos dois segmentos que ele toca.

Observe o Código 9.1. Os **for**s das linhas 3 e 4 iteram por todos os pontos dos polígonos e os armazenam no vetor *points*. A linha 6 ordena esses pontos. Dessa forma o **for** da linha 10 itera pelos pontos da esquerda para a direita. Usamos uma árvore rubro-negra parcialmente persistente, com a mesma API que a do Capítulo 8.

Os polígonos são dados em sentido anti-horário, como na Figura 9.3, então se o segmento vai da esquerda para a direita, os pontos imediatamente acima dele pertencem ao polígono (como o segmento de 1 para 2), e se vai da direita para a esquerda, então tais pontos não pertencem ao polígono (como o segmento de 3 para 4). Os **if**s das linhas 11-18 então adicionam ou removem os segmentos da ABB, dependendo se estamos analisando a ponta direita ou esquerda do segmento, e

Código 9.1: Preprocessamento para localização de ponto

```
1:  function PREPROCESS({P_1, ..., P_k})
2:      points = {}                                              ▷ Vetor vazio
3:      for P_i ∈ {P_1, ..., P_k} :
4:          for P_i^j ∈ P_i :
5:              points.ADD(P_i^j)
6:      Ordene points de forma que points[i] ≤ points[j] se i < j.   ▷ O(n lg n)
7:      rbt = Árvore rubro-negra parcialmente persistente do Capítulo 8.
8:      slabs = {}
9:      slabs.ADD(((−∞, 0), rbt.current))
10:     for P_i^j ∈ points :
11:         if P_i^{j−1} < P_i^j :          ▷ Segmento (j − 1, j) vai para a direita
12:             rbt.REMOVE(SEGMENT(i, P_i^{j−1}, P_i^j, true))          ▷ O(lg n)
13:         else
14:             rbt.INSERT(SEGMENT(i, P_i^j, P_i^{j−1}, false))         ▷ O(lg n)
15:         if P_i^{j+1} > P_i^j :          ▷ Segmento (j, j + 1) vai para a direita
16:             rbt.INSERT(SEGMENT(i, P_i^j, P_i^{j+1}, true))          ▷ O(lg n)
17:         else
18:             rbt.REMOVE(SEGMENT(i, P_i^{j+1}, P_i^j, false))         ▷ O(lg n)
19:         slabs.ADD((P_i^j, rbt.current))
```

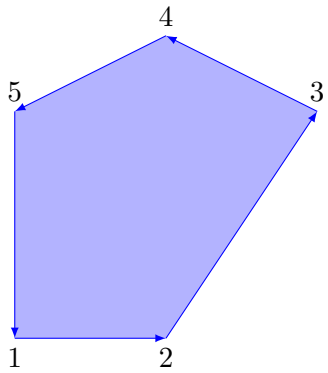

Figura 9.3: Exemplo do polígono dado em sentido anti-horário.

calculando o campo *top* de acordo se o segmento vai para a esquerda ou direita.

A ordenação dos segmentos, que está implicitamente sendo usada pela ABB, pode ser feita usando produto cruzado de vetores.

Por fim, a lista *slabs* é usada para armazenar a correspondência entre faixas e versões da ABB. Ela armazena, em ordem, o ponto correspondente a cada faixa e a versão da ABB que a representa (dada pelo campo *current* da ABB).

A complexidade desse código é $\mathcal{O}(n \lg n)$, já que realizamos $\mathcal{O}(n)$ adições e remoções à ABB, cada uma custando $\mathcal{O}(\lg n)$, e a ordenação também consome tempo $\mathcal{O}(n \lg n)$. Todos os passos que têm complexidade diferente de constante estão explicitados no Código 9.1.

9.5 Consulta

Para determinar a qual polígono pertence um dado ponto, precisamos determinar a qual faixa este pertence, e o segmento diretamente abaixo do ponto nesta faixa. Observe o Código 9.2.

Código 9.2: Respondendo consulta

```
1:  function WhichPolygon(p)
2:      Determine o último par (q, current) tal que q ≤ p.        ▷ Busca binária O(lg n)
3:      u = rbt.roots[current]
4:      below = null
5:      while u ≠ null :
            ▷ Se p está acima do segmento
6:          if Cross(u.value.to − u.value.from, p − u.value.from) ≥ 0 :
7:              below = u.value
8:              u = rbt.Child(u, 1, current)
9:          else
10:             u = rbt.Child(u, 0, current)
11:     if below = null :
12:         return −1
13:     else if not below.top or Cross(below.to − below.from, p − below.from) = 0 :
14:         return below.polygon
15:     else
16:         return −1
```

A linha 2 determina a faixa a qual pertence p, usando busca binária na lista *slabs*. Nas linhas 3-10, buscamos o segmento imediatamente abaixo do ponto p. Para isso usamos as funções apresentadas no Capítulo 8 e a função Cross, que calcula o produto cruzado de dois vetores, cujo sinal é usado para determinar se o ponto está acima do segmento. O procedimento é análogo ao procedimento de Floor, descrito por Sedgewick e Wayne [13].

Com o segmento encontrado, podemos determinar o polígono a qual p pertence, nas linhas 11-16. Se não existe segmento abaixo de p ou ele está acima de um segmento que era da parte "de cima" do polígono, então p não está em nenhum polígono. Caso contrário, o polígono é dado pelo segmento.

A complexidade da consulta é $\mathcal{O}(\lg n)$, já que tanto a busca binária quanto a busca na ABB têm essa complexidade.

Conclusão

A dissertação apresenta persistência em estruturas de dados, mostrando vários resultados dessa área, com atenção especial para a implementação destas estruturas de forma prática. Todas as estruturas apresentadas foram implementadas em C++, e as implementações podem ser acessadas pelo repositório `yancouto/mestrado` no GitHub. A documentação dessas implementações está disponível em `yancouto.github.io/mestrado`.

As implementações de deque apresentadas mostram como reduzir a complexidade pode complicar bastante a teoria e implementação de uma estrutura. Como indicado na Tabela 6.2, para reduzir a complexidade de tempo de duas operações da deque, de logarítmico pra constante, foi necessário apresentar toda a teoria e código complicados do Capítulo 6; e, na prática, com limites razoáveis para os computadores dos dias de hoje, esta solução fica mais lenta.

Apresentamos algumas técnicas gerais para tornar algumas classes de estruturas de dados em persistentes, e aplicamos algumas destas a estruturas conhecidas: apresentamos uma deque persistente que usar implementação funcional e uma árvore rubro-negra parcialmente persistente que usa node copying.

Por último, apresentamos uma aplicação de estruturas persistentes, a solução do problema de localização de ponto. É possível generalizar este tipo de solução, na qual fazemos uma linha de varredura com alguma estrutura de dados de forma offline, já que é possível utilizar uma versão persistente da estrutura para respondermos as consultas de forma online.

Bibliografia

[1] M.A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321:5–12, 2004.

[2] R. Cole. Searching and storing similar lists. *Journal of Algorithms*, 7(2):202 – 220, 1986.

[3] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3rd edition, 2001.

[4] D. Dobkin and R.J. Lipton. Multidimensional searching problems. *SIAM Journal on Computing*, 5, 06 1976.

[5] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[6] R.T. Hood and R.C. Melville. Real time queue operations in pure LISP. Technical report, 1980.

[7] H. Kaplan. *Handbook on Data Structures and Applications*, chapter 31, Persistent Data Structures. CRC Press, 2004. 27 pp.

[8] H. Kaplan and R.E. Tarjan. Purely functional, real-time deques with catenation. *J. ACM*, 46(5):577–603, September 1999.

[9] E.W. Myers. *AVL DAGs*. University of Arizona, Department of Computer Science, 1982.

[10] E.W. Myers. An applicative random-access stack. *Information Processing Letters*, pages 241–248, 1983.

[11] E.W. Myers. Efficient applicative data types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 66–75. ACM, 1984.

[12] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.

[13] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.