

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, dezembro de 2018

Estruturas de dados persistentes

Este exemplar corresponde à redação final da dissertação devidamente corrigida, defendida por Yan Soares Couto e aprovada pela Comissão Julgadora.

As opiniões, hipóteses e conclusões ou recomendações expressas neste material são de responsabilidade do autor e não necessariamente refletem a visão da FAPESP.

Processo 2017/05481-8
Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)

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.

Estruturas de dados (EDs) 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. Dizemos que, ao realizar uma operação de modificação, criamos uma nova versão da ED.

Uma ED é *parcialmente persistente* se permite apenas operações de acesso a versões anteriores e modificação apenas na versão mais nova, e *totalmente persistente* se também permite operações de modificação em todas as versões.

Esta dissertação apresenta a descrição e implementação de uma versão total ou parcialmente persistente de várias estruturas: pilhas, filas, deque e árvores rubro-negras. Também são discutidas técnicas gerais para tornar persistentes certas classes de estruturas de dados. Por fim, é apresentada uma solução ao problema de localização de ponto, que usa uma árvore de busca binária persistente.

Palavras-chave: estruturas de dados, persistência, árvores rubro-negras, 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.

Data structures (DSs) allow access and update operations; access operations only allow accessing the value of one or more fields of the DS, while update operations allow modifying the fields of the structure. We say that, whenever an update operation is done, a new version of the DS is created.

A DS is *partially persistent* if it allows access operations to previous versions of the structure and update operations only on the newest version, and *totally persistent* if it also allows update operations on all versions.

This dissertation presents the description and implementation of totally or partially persistent versions of several data structures: stacks, queues, deques, and red-black trees. General techniques to make certain classes of DSs persistent are also discussed. At last, a solution to the point location problem, using a persistent binary search tree, is presented.

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

Sumário

Introdução	1
I Preliminares	3
1 Ancestrais em árvores	4
1.1 Introdução	4
1.2 Potências de funções	4
1.3 Ancestral de nível online para árvores	6
1.4 Cálculo do ancestral comum mais profundo	7
2 Ancestrais com representação skew binary	10
2.1 Definição e propriedades	10
2.2 Jump pointers	12
2.3 Cálculo dos jump pointers	14
2.4 Ancestral comum mais profundo	15
II Persistência	17
3 Pilhas e filas de acesso aleatório	18
3.1 Pilhas persistentes	18
3.2 Filas persistentes	22
4 Deque com LA e LCA	24
4.1 Representação e visão geral	25
4.2 Acesso e inserção	26
4.3 Remoção	27
4.4 Acesso a outros elementos	28
5 Deque recursiva	30
5.1 Representação	30
5.2 Operações de acesso	31
5.3 Operações de modificação	32
5.4 Acesso a outros elementos	34

6	Deque de Kaplan e Tarjan	36
6.1	Contadores binários	36
6.2	Visão geral	38
6.3	Regularidade e operações	39
6.4	Pilha de pilhas, e implementações funcionais	40
6.5	Representação	43
6.6	Procedimento FIX detalhado	43
6.7	Implementação de FIXDEQUES	44
6.8	Implementação de FIX	45
6.9	Implementação das operações	47
7	Técnicas gerais	49
7.1	Modelo de computação	49
7.2	Offline	50
7.3	Implementação funcional	52
7.4	Fat node	54
7.5	Node copying	55
8	Árvore rubro-negra	62
8.1	Definições	63
8.2	Implementação da persistência	63
8.3	Operações de acesso	64
8.4	Modificação de um campo	65
8.5	Inserção em ABB	65
8.6	Inserção em rubro-negra	67
8.7	Remoção em ABB	71
8.8	Remoção em rubro-negra	73
9	Localização de ponto	79
9.1	Solução ingênua	79
9.2	Partição do plano em faixas	80
9.3	Conversão da solução offline em online	81
9.4	Pré-processamento	81
9.5	Consulta	83
	Conclusão	84

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, Sarnak, 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 as soluções para os problemas de ancestral comum mais profundo e ancestral de nível, usadas como caixa preta nos Capítulos 3 e 4. Um leitor com algum conhecimento prévio destes tópicos pode optar por iniciar a leitura desta dissertação a partir da Parte II, consultando a Parte I se necessário.

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 deque.
- O Capítulo 7 apresenta técnicas gerais para tornar persistentes certas classes de estruturas de dados. O Capítulo 8 aplica uma destas técnicas à árvore rubro-negra.
- O Capítulo 9 apresenta uma aplicação da estrutura apresentada no Capítulo 8 para resolver um problema de geometria computacional conhecido como o problema de localização de ponto.

INTRODUÇÃO

Todas as estruturas apresentadas foram implementadas em C++, e as implementações podem ser acessadas pelo repositório [yancouto/mestrado](https://github.com/yancouto/mestrado) no GitHub. A documentação dessas implementações está disponível em yancouto.github.io/mestrado.

Parte I

Preliminares

Capítulo 1

Ancestrais em árvores

1.1 Introdução

Seja $T := (V, E)$ 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, assumimos que $V = \{1, 2, \dots, |V|\}$. 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. Em particular, u é o 0-ésimo ancestral de u . Dizemos que o primeiro ancestral de u , se $u \neq r$, é o *pai* de u . Definimos a função $\text{Parent} : V \rightarrow V$ tal que $\text{Parent}(u)$ é 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 denotada por $D(u)$ e é 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 $k \in \mathbb{N}$ tal que $k \leq D(u)$, o k -ésimo ancestral de u , ou seja, o problema de avaliar a função

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

O problema do Primeiro Ancestral Comum (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

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

1.2 Potências de funções

Vamos considerar o problema, um pouco mais genérico, de, dada uma função $f : [n] \rightarrow [n]$, 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 $\langle \mathcal{O}(p(n, m)), \mathcal{O}(q(n, m)) \rangle$.

1.2.1 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 $\langle \mathcal{O}(nm), \mathcal{O}(1) \rangle$.

1.2.2 Potências de dois

As soluções apresentadas funcionam da seguinte maneira: escolhe-se uma base (a_1, \dots, 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 $k = a_{b_1} + a_{b_2} + \dots + a_{b_y}$, e após isso calcula-se

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

No primeiro exemplo da Subseção 1.2.1 escolhemos como base apenas (1), e o tempo de consulta foi grande, enquanto no segundo exemplo escolhemos $(1, \dots, m)$, 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 à sua representação binária), e existem apenas $\lfloor \lg m \rfloor$ potências de 2 entre 1 e m . Portanto, podemos escolher a base $(1, 2, 4, \dots, 2^{\lfloor \lg m \rfloor})$. Para fazer o pré-processamento, utilizaremos programação dinâmica preenchendo uma matriz M tal que $M[k][i] := f^{2^k}(i)$ para todo $i \in [n]$ e $0 \leq k \leq \lfloor \lg m \rfloor$. Sabemos que $f^x(f^x(i)) = f^{2^x}(i)$, 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 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 . \square

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 procedimento 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  $\lfloor \lg m \rfloor$  :
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( $k, i$ ) ▷  $k \leq m$  e  $i \leq n$ 
8:   for  $x = \lfloor \lg m \rfloor$  down to 0 :
9:     if  $2^x \leq k$  :
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 $\text{Parent}^k(u)$ para $k < D(u)$, logo usamos $n = m = |V|$. Estes algoritmos, porém, supõem que a função é conhecida de antemão, para realizar o pré-processamento, ou seja, a função Parent deve ser totalmente conhecida de antemão (e por conseguinte a árvore).

Na Subseção 1.2.2, vimos que $M[k][u] = M[k-1][M[k-1][u]]$ pode ser calculada usando programação dinâmica pois depende de índices menores da primeira coordenada. 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 $0 \leq k < \lfloor \lg D(u) \rfloor$ basta que os valores de M já estejam calculados para todos os ancestrais de u , e então calculamos $M[k][u]$ 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, com novas folhas podendo ser adicionadas à árvore. Para melhorar a notação neste caso, usamos em cada nó um vetor *jump* com os valores de M , ou seja, $u.\text{jump}[k] = M[k][u] = \text{Parent}^{2^k}(u)$. Assim que uma folha u for

adicionada, podemos calcular os valores de $u.jump$, em tempo e espaço $\mathcal{O}(\lfloor \lg D(u) \rfloor)$. 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:    $u.jump[0] = \text{Parent}(u)$ 
3:   for  $k = 1$  to  $\lfloor \lg D(u) \rfloor$  :
4:      $u.jump[k] = u.jump[k-1].jump[k-1]$ 
  ▷ Devolve  $\text{LA}(k, u)$ ; deve ser chamada após ADDLEAF( $u$ ).
5: function LEVELANCESTOR( $k, u$ )
6:   for  $x = \lfloor \lg k \rfloor$  down to  $0$  :
7:     if  $2^x \leq k$  :
8:        $k = k - 2^x$ 
9:        $u = u.jump[x]$ 
10:  return  $u$ 

```

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 $\mathcal{O}(\lg D(u))$ e a operação LEVELANCESTOR(k, u) consome tempo $\mathcal{O}(\lg k) = \mathcal{O}(\lg D(u))$. Note que cada nó u deve armazenar um vetor de tamanho $\lfloor \lg D(u) \rfloor$. 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.

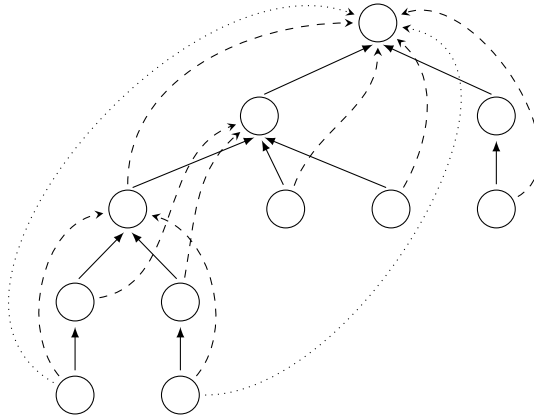


Figura 1.1: Exemplo de árvore com jump pointers. As arestas cheias são as arestas de pai ($jump[0]$), as arestas tracejadas pulam dois níveis ($jump[1]$), e as arestas pontilhadas pulam quatro níveis ($jump[2]$).

1.4 Cálculo do ancestral comum mais profundo

Sejam $u, v \in V$ e $c \in V$ o ancestral comum mais profundo de u e v . Assuma, sem perda de generalidade, que $D(v) \geq D(u)$. É óbvio que $D(c) \leq D(u)$, logo podemos “nivelar” os vértices u e v , ou seja, trocar v por $\text{LA}(D(v) - D(u), v)$. Assim, podemos supor que $D(u) = D(v)$ e, para encontrar o ancestral comum mais profundo de u e v , devemos encontrar o menor k^* tal

que $\text{LA}(k^*, u) = \text{LA}(k^*, v)$. Note que se $k < D(u) - 1$ e $\text{LA}(k, u) = \text{LA}(k, v)$, então

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

Isso permitirá a utilização dos Jump Pointers para determinar tal k^* mínimo.

Se $x < k^*$ então $\text{LA}(x, u) \neq \text{LA}(x, v)$, ou seja, temos uma forma de checar se $x < k^*$ sem conhecer k^* . 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. Como temos jump pointers para as potências de dois para todos os nós, conseguimos determinar a decomposição em potências de dois distintas de $k^* - 1$ (já que conseguimos testar $x \leq k^* - 1$ e não $x \leq k^*$), e de forma similar à função `LEVELANCESTOR` do Código 1.2, podemos determinar o $(k^* - 1)$ -ésimo ancestral de u e v em tempo $\mathcal{O}(\lg D(u))$.

Código 1.3: Primeiro Ancestral Comum usando Jump Pointers.

```

1: function LOWESTCOMMONANCESTOR( $u, v$ )
2:   if  $D(u) > D(v)$  :
3:     |  $u, v = v, u$                                      ▷ Garante que  $D(u) \leq D(v)$ .
4:      $v = \text{LEVELANCESTOR}(D(v) - D(u), v)$              ▷ Nivelava  $v$ .
5:     if  $u = v$  :
6:       | return  $u$ 
7:     for  $i = \lfloor \lg D(u) \rfloor$  down to 0 :
8:       | if  $u.\text{jump}[i] \neq v.\text{jump}[i]$  :
9:         |    $u = u.\text{jump}[i]$ 
10:        |    $v = v.\text{jump}[i]$ 
11:       |   ▷  $u$  é agora o filho do LCA de  $u$  e  $v$ .
12:     return  $\text{Parent}(u)$ 

```

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

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

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

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

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 $d := D(u) - D(c)$. Sabemos então que $0 < d \leq 2^{i+1}$. Note que $d \leq 2^i$ se e somente se $\text{LA}(2^i, u) = \text{LA}(2^i, v)$, 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 $\text{LA}(2^i, u)$ e v por $\text{LA}(2^i, v)$. Seja d' o novo valor de $D(u) - D(c)$. A profundidade de u e v diminui de 2^i , logo o valor de

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

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

□

Portanto, ao final da última iteração, o invariante vale para $i = -1$, ou seja, vale que $0 < D(u) - D(c) \leq 2^0 = 1$, 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.

Função	Tempo/Espaço
<u>ADDLEAF</u> (u)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
<u>LEVELANCESTOR</u> (k, u)	$\mathcal{O}(\lg n)$
<u>LOWESTCOMMONANCESTOR</u> (u, v)	$\mathcal{O}(\lg n)$

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

Capítulo 2

Ancestrais 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. Esta solução foi dada inicialmente por Myers [10]. No final do capítulo, apresentamos também uma solução para o problema de Primeiro Ancestral Comum utilizando a mesma técnica.

Como lidaremos com representações numéricas em grande detalhe, neste capítulo usaremos também notação de cadeias, na qual 21^5 representa a cadeia 211111 e não o número 4084101.

2.1 Definição e propriedades

Um número *skew-binary* de tamanho n é uma cadeia $a = a_n a_{n-1} \cdots a_1$ tal que $a_i \in \{0, 1, 2\}$ e $a_n \neq 0$. Denotamos o tamanho de a por $|a| = n$. O valor de tal número é $V(a) := \sum_{i=1}^n a_i(2^i - 1)$. Note que múltiplas cadeias podem ter o mesmo valor, por exemplo, tanto 21 quanto 100 têm valor 7.

Para todo a tal que $V(a) \neq 0$, defina $NZ(a) := \min\{i \in [n] : a_i \neq 0\}$, 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, $a_i = 2$ implica que $i = NZ(a)$. Seja *CSB* o conjunto de todos os números skew-binary canônicos.

Esta igualdade será útil na prova dos lemas:

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

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

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 $V(a) \geq V(b) = 2^n - 1$.

Note que

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

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

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 natural tem uma representação única em CSB. Equivalentemente, $V : \text{CSB} \rightarrow \mathbb{N}$ é uma função bijetora.*

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

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

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

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

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

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

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

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

Como cada número natural tem uma representação única em CSB, podemos definir uma função $R : \mathbb{N} \rightarrow \text{CSB}$ tal que $V(R(x)) = x$ para todo $x \in \mathbb{N}$, ou seja, $R(x)$ é a representação skew-binary canônica de x .

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

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

Caso contrário, como o único dígito em a que pode ser 2 é o $\text{NZ}(a)$ -ésimo dígito, temos que $b := a_n \cdots a_{\text{NZ}(a)+1} (a_{\text{NZ}(a)} - 1) 20^{\text{NZ}(a)-2} \in \text{CSB}$. Além disso,

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

□

O lema mostra que subtração por 1 no valor de um número 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 $\text{LA}(k, u)$, temos um nó u de profundidade $D(u)$ e queremos determinar seu ancestral v de profundidade $D(v) = D(u) - k$. Na solução apresentada na Subseção 1.2.2, cada nó tinha um ponteiro para seu 2^x -ancestral, para todo $x \in [\lg D(u)]$, 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 1.2.2, que tenta transformar $D(u)$ em $D(v)$, 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 a partir de um número x positivo podemos pular para os números $x - 1$ ou $J(x)$, onde $J(x) := x - (2^{\text{NZ}(R(x))} - 1)$, ou seja, se considerarmos $R(x)$, a representação skew-binary canônica de x , $J(x)$ consiste em diminuir de um o dígito não nulo menos significativo de $R(x)$. Por exemplo, se $x = 13$, então $R(x) = 120$ e $J(x) = V(110) = 10$. A Figura 2.1 dá os valores de R e J para números pequenos.

x	$R(x)$	$J(x)$
0	0	0
1	1	0
2	2	1
3	10	1
4	11	3
5	12	4
6	20	3
7	100	0
8	101	7
9	102	8
10	110	7
11	111	10
12	112	11
13	120	10
14	200	7
15	1000	0

Figura 2.1: Valores das funções R e J para valores pequenos.

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

Código 2.1: Transformação de x em y usando $x - 1$ e $J(x)$.

```

1: while  $x \neq y$  :
2:   |   if  $J(x) \geq y$  :
3:     |    $x = J(x)$ 
4:   |   else
5:     |    $x = x - 1$ 

```

O algoritmo é guloso no sentido que sempre escolhe usar J quando possível (e $J(x) \leq x - 1$). 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 $a := R(x)$, $b := R(y)$ e $n := |a|$. Se $|a| > |b|$, aumente b adicionando 0s à esquerda. Seja $i^* = \max\{i \in [n] : a_i \neq b_i\}$, e escreva $a = \alpha a_{i^*} \beta$ e $b = \alpha b_{i^*} \gamma$. Pelo Lema 2.1, temos que $a_{i^*} > b_{i^*}$ e que $V(c) > V(b)$, onde $c := b_n \cdots b_{i^*+1} (b_{i^*} + 1) 0^{i^*-1}$. Note que $c_i \leq a_i$ para todo $i \in [n]$, e temos que $\text{NZ}(a) \leq i^*$. Se $\text{NZ}(a) < i^*$ vale que $a_{\text{NZ}(a)} > 0 = c_{\text{NZ}(a)}$, e se $\text{NZ}(a) = i^*$ vale que $a_{\text{NZ}(a)} = a_{i^*} \geq b_{i^*} + 1 = c_{i^*}$. Portanto $a_{\text{NZ}(a)} \geq c_{\text{NZ}(a)}$, com igualdade apenas se $a = c$. Logo se $a \neq c$ podemos diminuir $a_{\text{NZ}(a)}$ 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 $\sum_{i=1}^{i^*} a_i \leq i^* + 1$ iterações, já que cada dígito é no máximo 1, a menos de um destes, que pode ser 2.

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

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 $i > 1$ e a hipótese vale para valores menores que i . Se $\text{NZ}(b) \geq i$, então b tem um sufixo de pelo menos $i - 1$ valores 0. Ou seja $a = b_n \cdots b_{i+1} (b_i + 1) b_{i-1} \cdots b_1$, e então se diminuirmos 1 de a_i temos b , ou seja, $J(V(a)) = V(b)$, e o algoritmo termina em uma iteração. Caso contrário, $J(V(a)) < V(b)$, e então $x = x - 1$ é executado. Pelo Lema 2.3, $R(x - 1) = b_n \cdots b_i 20^{i-2}$, e é claro que $V(20^{i-2}) \geq V(b_{i-1} \cdots b_1)$. 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 $x - 1$ e $i - 1$) e temos que o algoritmo gasta no máximo $2(i - 1) + 1 < 2i$ iterações;
- se $b_{i-1} = 0$, então $J(x - 1) > y$ e $R(J(x - 1)) = b_n \cdots b_i 10^{i-2}$, logo podemos aplicar a hipótese de indução (para $J(x - 1)$ e $i - 1$) e temos que o algoritmo gasta no máximo $2(i - 1) + 2 = 2i$ iterações.

□

A Figura 2.2 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. O resto corresponde à segunda parte.

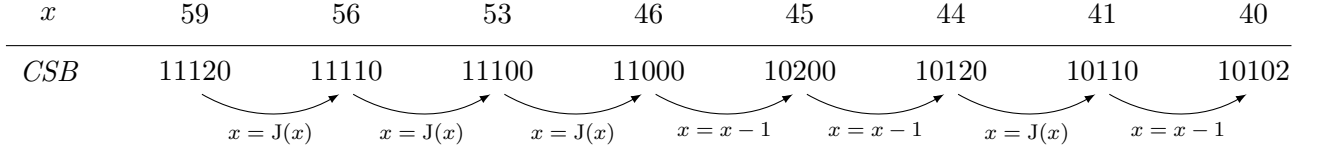


Figura 2.2: Exemplo do algoritmo do Código 2.1 aplicado sobre $x = 59$ e $y = 40$.

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 $J(D(u))$. 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 $\mathcal{O}(\lg D(u))$ segue diretamente do Teorema 2.4.

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

```

1: function LEVELANCESTOR( $k, u$ )
2:    $y = D(u) - k$ 
3:   while  $D(u) \neq y$  :
4:     if  $D(u.jump) \geq y$  :
5:        $u = u.jump$ 
6:     else
7:        $u = \text{Parent}(u)$ 
8:   return  $u$ 

```

2.3 Cálculo dos jump pointers

A seção anterior apresentou uma solução para o problema do Ancestral de Nível que consome tempo logarítmico, porém nela assumimos que cada nó u tinha um campo $u.jump$ com seu $J(D(u))$ -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 2, ou seja, se $a_{\text{NZ}(a)} \neq 2$, então $J(V(a) + 1) = V(a)$. Caso contrário, $J(V(a) + 1) = J(J(V(a)))$.*

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

Caso contrário, $a_{\text{NZ}(a)} = 2$. Considere $b := a_n \cdots a_{\text{NZ}(a)+2}(a_{\text{NZ}(a)+1} + 1)0^{\text{NZ}(a)}$. Pelo Lema 2.3 vale que $R(V(b) - 1) = a$, logo $V(b) = V(a) + 1$. Note que

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

Portanto, $J(V(a) + 1) = J(J(V(a)))$. □

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

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

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

Demonstração. Seja $b := R(J(V(a)))$. Lembre que $J(V(a)) = V(a) - (2^{\text{NZ}(a)} - 1)$.

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

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

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

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

□

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 .

Código 2.3: Adição de uma folha à árvore com raiz r .

```

1: function ADDLEAF( $u$ )
2:    $v = \text{Parent}(u)$ 
3:   if  $v.\text{jump} \neq r$  and  $D(v) - D(v.\text{jump}) = D(v.\text{jump}) - D(v.\text{jump}.\text{jump})$  :
4:      $u.\text{jump} = v.\text{jump}.\text{jump}$ 
5:   else
6:      $u.\text{jump} = v$ 

```

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 condições 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.

2.4 Ancestral comum mais profundo

Para encontrar 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 $J(x) \geq y$. Sabemos que $u.\text{jump} = v.\text{jump}$ se e somente se $D(u.\text{jump}) > D(c)$, 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: Primeiro Ancestral Comum usando representação skew-binary.

```

1: function LOWESTCOMMONANCESTOR( $u, v$ )
2:   if  $D(u) > D(v)$  :
3:     |    $u, v = v, u$                                      ▷ Garante que  $D(u) \leq D(v)$ .
4:     |    $v = \underline{\text{LEVELANCESTOR}}(D(v) - D(u), v)$            ▷ Nivelava  $v$ .
5:     |   if  $u = v$  :
6:       |   return  $u$ 
7:     |   while  $\text{Parent}(u) \neq \text{Parent}(v)$  :
8:       |   if  $u.\text{jump} \neq v.\text{jump}$  :
9:         |   |    $u = u.\text{jump}$ 
10:        |   |    $v = v.\text{jump}$ 
11:       |   else
12:         |   |    $u = \text{Parent}(u)$ 
13:         |   |    $v = \text{Parent}(v)$ 
14:       |   ▷  $u$  é agora o filho do LCA de  $u$  e  $v$ .
14:     |   return  $\text{Parent}(u)$ 

```

Função	Representação binária	Skew-binary
<u>ADDLEAF</u> (u)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>LEVELANCESTOR</u> (k, u)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$
<u>LOWESTCOMMONANCESTOR</u> (u, v)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 2.1: Comparação do consumo de tempo e espaço das soluções dos Capítulos 1 e 2, onde n é o tamanho da árvore.

A Tabela 2.1 compara o consumo de tempo das implementações de Ancestral de Nível e Primeiro Ancestral Comum apresentadas nesse trabalho.

Parte II

Persistência

Capítulo 3

Pilhas e filas de acesso aleatório

Em nosso pseudocódigo, usaremos estruturas de dados como objetos. As operações aplicadas sobre uma determinada ED são funções que recebem essa ED como argumento (além de possíveis argumentos adicionais). Sempre existe uma operação que devolve uma nova instância dessa ED vazia. As operações são o meio com o qual o cliente lida com a ED. Chamamos de efêmera uma implementação usual de uma ED, que não é necessariamente persistente.

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 disponível, ou seja, continue podendo ser usada tanto para operações de acesso como de modificação. A maneira como a versão anterior está armazenada pode mudar, mas a resposta para qualquer operação sobre ela deve ser a mesma. Dizemos que a versão atual da ED é a última versão criada por alguma operação de modificação.

Como aquecimento, a seguir apresentamos implementações persistentes de duas estruturas de dados simples: pilhas e filas. A descrição de pilhas persistentes, incluindo a operação K-TH, foi feita por Myers [10].

3.1 Pilhas persistentes

Pilhas são uma das estruturas de dados mais simples. Usualmente as seguintes operações estão disponíveis para se manipular uma pilha:

- STACK()
Devolve uma pilha vazia.
- PUSH(p, x)
Devolve uma cópia da pilha p com o valor x inserido no seu topo.
- POP(p)
Devolve uma cópia da pilha p com o elemento do seu topo removido.
- SIZE(p)
Devolve o número de elementos na pilha p .

- $\underline{\text{TOP}}(p)$

Devolve o elemento do topo da pilha p .

Uma pilha é de *acesso aleatório* se permite acesso a qualquer elemento seu, não apenas a seu topo. Para que os acessos a uma tal pilha estejam bem definidos, consideramos seus elementos na ordem em que foram inseridos. Dessa maneira, o topo da pilha é o seu último elemento e a seguinte operação está também disponível:

- $\underline{\text{K-TH}}(p, k)$

Devolve o k -ésimo elemento da pilha p .

Observe que as operações $\underline{\text{PUSH}}$ e $\underline{\text{POP}}$ são de modificação enquanto que $\underline{\text{TOP}}$, $\underline{\text{K-TH}}$ e $\underline{\text{SIZE}}$ são operações de acesso. No resto desta seção descreveremos a implementação de uma pilha de acesso aleatório totalmente persistente. Para deixar clara a notação, operações são SUBLINHADAS e funções auxiliares (que usamos para nos ajudar a implementar as operações) não são.

3.1.1 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.

Para que as operações tenham implementações eficientes, os elementos da pilha são armazenados em uma lista ligada na ordem inversa, ou seja, o último elemento da pilha (seu topo) é armazenado no primeiro nó da lista ligada. A pilha é representada por um ponteiro para esse primeiro nó da lista ligada, ou para **null**, se a pilha está vazia.

Para realizar a operação $\underline{\text{PUSH}}(p, x)$, basta criar um novo nó com o valor x e inseri-lo no início da lista ligada. O novo nó passa a ser o primeiro nó da lista ligada. Na implementação de uma pilha efêmera, para realizar a operação $\underline{\text{POP}}(p)$, basta atualizar o ponteiro da pilha para o segundo nó da lista. O primeiro nó pode ser devolvido ou descartado.

Em princípio, não é necessário alterar os campos dos nós já criados. Em particular, poderíamos manter todos os nós já criados, sem alterá-los. Dessa forma, se guardarmos em p_i um ponteiro para o primeiro nó da lista ligada após a i -ésima operação de modificação, o ponteiro p_i nos daria acesso a i -ésima versão da pilha. De fato, os nós acessíveis a partir de p_i não mudarão mesmo após futuras operações. Portanto é possível realizar operações de acesso e modificação em versões anteriores da pilha, resultando em uma implementação persistente dessa.

Implementações de estruturas como esta, nas quais operações de modificação não modificam 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, é fácil obter uma implementação persistente para tais estruturas. 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. Considere que `new NODE(x, nx, sz)` cria um novo nó com campos $val = x$, $next = nx$ e $size = sz$. Note que ignoramos a operação $\underline{K-TH}(p, k)$ por ora, mas voltaremos a discuti-la na Subseção 3.1.3.

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

```

3.1.2 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 $\underline{TOP}(p)$.

$p_0 = \underline{STACK}()$ $p_1 = \underline{PUSH}(p_0, 5)$ $p_2 = \underline{PUSH}(p_1, 7)$ $p_3 = \underline{PUSH}(p_2, 6)$ $p_4 = \underline{POP}(p_2)$ $\underline{TOP}(p_3)$ $p_5 = \underline{PUSH}(p_4, 9)$ $\underline{TOP}(p_4)$ $p_6 = \underline{PUSH}(p_0, 5)$	$p_0 :$ $p_1 : 5$ $p_2 : 5 7$ $p_3 : 5 7 6$ $p_4 : 5$ Devolve 6 $p_5 : 5 9$ Devolve 5 $p_6 : 5$
---	---

Exemplo 3.1: Exemplo de uso de uma pilha persistente.

Em uma pilha efêmera, adicionamos apenas elementos no início 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 em direção à raiz, considerando que a raiz é o valor `null` (pilha vazia). Note que, como discutido anteriormente, a partir de cada nó podemos apenas acessar os nós do caminho deste nó até a raiz, utilizando 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, \dots, 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.

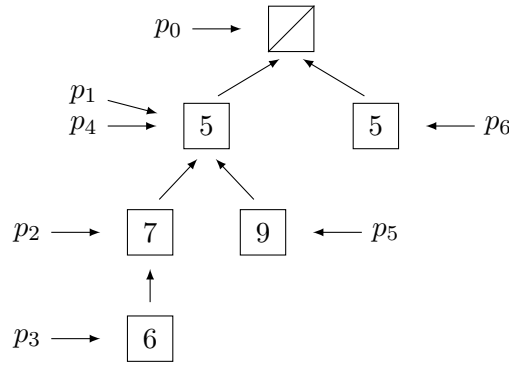


Figura 3.1: Arborescência criada pela sequência de operações do Exemplo 3.1. A raiz na figura (o nó cortado) é a representação do valor **null**.

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 .

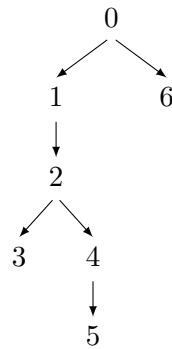


Figura 3.2: Árvore de versões associada à sequência de operações do Exemplo 3.1.

3.1.3 Acesso a outros elementos

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

Lembre-se que p é um nó de uma arborescência, e representa o último elemento da pilha. Logo $\text{K-TH}(p, k)$ 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. Nesses capítulos foi discutido como encontrar ancestrais de níveis usando as operações $\text{ADDLEAF}(u)$ e $\text{LEVELANCESTOR}(k, u)$. Note que apresentamos essas operações de forma que folhas novas possam ser adicionadas de forma online, o que é necessário neste problema, já que adicionamos uma folha quando aplicamos uma operação PUSH à pilha.

Se assumimos que a função **new** $\text{NODE}(x, nx, sz)$ também chama a operação $\text{ADDLEAF}(u)$ para o novo nó criado, com $\text{Parent}(u) = nx$, então o Código 3.2 implementa a operação $\text{K-TH}(p, k)$, e está trivialmente correto.

Código 3.2: Implementação de $\text{K-TH}(p, k)$ usando Ancestral de Nível como caixa preta.

```

1: function  $\text{K-TH}(p, k)$ 
2:   return  $\text{LEVELANCESTOR}(p.size - k, p).val$ 

```

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

Operação	Representação Binária	Skew-Binary
$\text{STACK}()$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$
$\text{PUSH}(p, x)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
$\text{POP}(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\text{TOP}(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\text{SIZE}(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\text{K-TH}(p, k)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

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 estrutura da pilha persistente.

3.2 Filas persistentes

Nesta seção apresentaremos uma implementação persistente bastante natural que desenvolvemos para filas, usando uma pilha persistente. A ideia dessa implementação surgiu quando elaboramos 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.

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

- $\text{QUEUE}()$
Devolve uma fila vazia.
- $\text{ENQUEUE}(q, x)$
Devolve uma cópia da fila q com o valor x inserido em seu fim.
- $\text{DEQUEUE}(q)$
Devolve uma cópia da fila q com seu primeiro elemento removido.
- $\text{SIZE}(q)$
Devolve o número de elementos na fila q .
- $\text{FIRST}(q)$
Devolve o primeiro elemento da fila q .
- $\text{K-TH}(q, k)$
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 $\text{K-TH}(q, k)$ e $\text{FIRST}(q)$. 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 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 início 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.

Operação	Representação Binária	Skew-Binary
<u>QUEUE</u> ()	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>ENQUEUE</u> (q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>DEQUEUE</u> (q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<u>SIZE</u> (q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<u>FIRST</u> (q)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$
<u>K-TH</u> (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 estrutura da fila persistente.

Capítulo 4

Deque com LA e LCA

Uma *fila com 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 *double ended queue*. Chama-se uma das extremidades da deque de início e outra de fim, embora ambas tenham o mesmo papel.

Uma deque de acesso aleatório admite as seguintes operações:

- DEQUE()
Devolve uma deque vazia.
- FRONT(d)
Devolve o primeiro elemento de d .
- BACK(d)
Devolve o último elemento de d .
- 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.
- 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), podemos simular uma pilha. Analogamente, usando apenas PUSHBACK(d, x), FRONT(d) e POPFRONT(d), podemos simular uma fila simples.

Na literatura existem outras implementações persistentes de deques, que serão apresentadas nos dois próximos capítulos. Neste capítulo, apresentaremos uma implementação persistente que

$(first_0, last_0) = \text{DEQUE}()$	\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.

desenvolvemos de deques. Ela se baseia nas implementações de pilhas e filas persistentes discutidas no Capítulo 3, e utiliza Ancestral de Nível e Primeiro Ancestral Comum, discutidos nos Capítulos 1 e 2.

Embora utilize os algoritmos para Ancestral de Nível e Primeiro Ancestral Comum, acreditamos que esta implementação seja mais simples que as 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 nós; e então é possível adicionar elementos ao final da pilha.

Na implementação de filas persistentes, descrita no mesmo capítulo, também foi discutido como simular a remoção de elementos do início da estrutura, guardando, junto ao nó indicado pela versão da fila, o número de elementos já removidos naquela versão. Dessa forma, os elementos de uma determinada versão da fila são um sufixo do caminho da raiz até o nó indicado pela versão. Essa organização, porém, não permite adicionar elementos no início da fila.

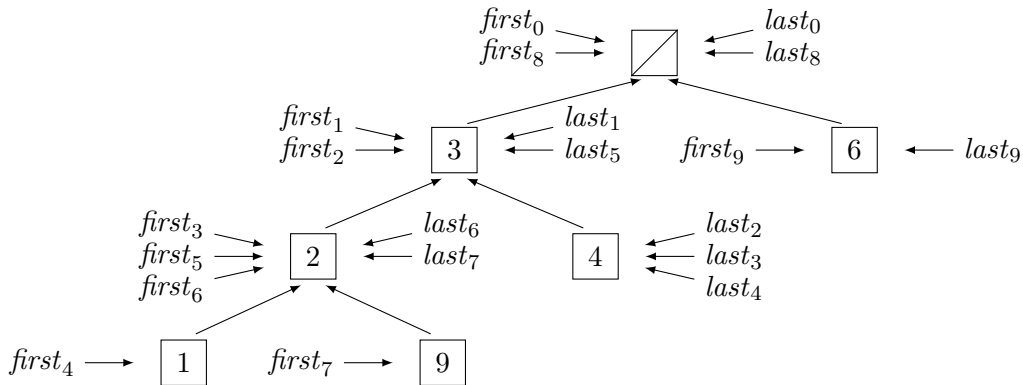


Figura 4.1: Arborescência criada pela sequência de operações do Exemplo 4.1. A raiz na figura (o nó cortado) é a representação do valor **null**.

Para a implementação de deques, ainda usaremos uma arborescência, mas cada versão da deque indicará dois nós: *first* e *last*. Os elementos dessa versão da deque serão os elementos no caminho entre *first* para *last* na árvore subjacente à arborescência (ou seja, na versão não dirigida da

arborescência). 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 de uma determinada versão da deque, criamos uma nova folha conectada ao *first* dessa versão e indicamos essa nova folha como o *first* da nova versão criada e o seu *last* é o mesmo da versão que foi modificada; para adicionar um elemento no final de uma determinada versão da deque, criamos uma nova folha conectada a *last* e definimos o *first* e *last* da nova versão analogamente. Criar folhas em uma árvore não muda os caminhos entres vértices já existentes, então as versões anteriores da deque continuam válidas, e o novo par $(first, last)$ representa a deque alterada com um elemento adicionado em seu início ou final. Isso funciona se a versão da deque em questão tem pelo menos um elemento antes da adição. Na Figura 4.1, ao inserir 1 no início da deque representada por $(first_3, last_3)$, obtemos a deque $(first_4, last_4)$.

Para remover um elemento do início de uma versão da deque indicada pelo par $(first, last)$, tomamos como *first* da nova versão o segundo elemento no caminho entre *first* e *last*; para remover um elemento do final dessa versão, tomamos como *last* o vizinho de *last* no caminho entre *first* e *last*. Isso funciona se a versão da deque em questão tem pelo menos dois elementos antes da remoção. 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 $(first_6, last_6)$.

É necessária uma representação especial para deque vazias, por exemplo, o par $(\mathbf{null}, \mathbf{null})$. Assim, ao remover um elemento de uma deque com um único elemento (ou seja, quando $first = last$), devolvemos o par $(\mathbf{null}, \mathbf{null})$; e para adicionar algum elemento à deque vazia basta criarmos um nó u com o valor desejado e devolvermos o par (u, u) . Dessa forma nossa estrutura é na verdade uma coleção de arborescências, ou podemos considerar que é uma arborescência em que \mathbf{null} representa a raiz, 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 $(first_8, last_8)$, onde o nó cortado representa \mathbf{null} , e ao adicionar 6 a essa deque obtemos $(first_9, last_9)$.

4.2 Acesso e inserção

Em cada nó da arborescência guardaremos: o pai do nó no campo *parent*; sua profundidade no campo *depth*; e seu valor associado, que é um dos valores armazenados na deque, no campo *val*. O código `new NODE(x, p, d)` cria um novo nó com valores x , p e d nos campos *val*, *parent* e *depth*, respectivamente, e também realiza o pré-processamento necessário para o algoritmo de Ancestral de Nível funcionar, ou seja, chama a função `ADDLEAF` para o novo nó, 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 inverter *first* e *last* no par associado a d), 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 de que os elementos de uma deque são os valores nos 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 FRONT(d)
4:   return d.first.val
5: function BACK(d)
6:   return d.last.val
7: function SWAP(d)
8:   return (d.last, d.first)
9: function PUSHFRONT(d, x)
10:  if d.first = null :
11:    u = new NODE(x, null, 1)
12:    return (u, u)
13:  else
14:    return (new NODE(x, d.first, d.first.depth + 1), d.last)
15: function PUSHBACK(d, x)
16:  return SWAP(PUSHFRONT(SWAP(d), x))

```

4.3 Remoção

Discutiremos como determinar o segundo elemento no caminho de *first* para *last*. Encontrar o penúltimo é encontrar o segundo no caminho de *last* para *first*, assim usaremos SWAP para não ter que tratar esse caso separadamente. Para uma árvore enraizada, 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*, ..., *mid*); assim, se *first* ≠ *mid*, temos que *first.parent* é o segundo elemento do caminho. Se *first* = *mid* então o caminho de *first* para *last* é (*first*, ..., *last.parent*, *last*). Este caminho tem comprimento *last.depth* − *first.depth*, e então o segundo nó deste caminho é o (*last.depth* − *first.depth* − 1)-ésimo ancestral de *last* na arborescência, considerando que o 0-ésimo ancestral de um nó é ele mesmo, o 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 LCA(*d.first*, *d.last*) = *d.first* para determinar se *d.first* é o ancestral comum mais profundo de *d.first* e *d.last*. Esta condição pode também ser verificada usando ape-

nas chamadas para LEVELANCESTOR. Se $first$ é o ancestral comum mais profundo de $first$ e $last$, então o caminho de $first$ para $last$ é $(first, \dots, last.parent, last)$, mas neste caso $d.first$ é o $(last.depth - first.depth)$ -ésimo ancestral de $last$. Portanto

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

se e somente se

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

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 , então 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:    $mid = \underline{LCA}(d.first, d.last)$ 
3:    $\ell_1 = d.first.depth - mid.depth$ 
4:    $\ell_2 = d.last.depth - mid.depth$ 
5:   if  $k - 1 \leq \ell_1$  :
6:     |   return LEVELANCESTOR( $k - 1, d.first$ )
7:   else
8:     |   return LEVELANCESTOR( $\ell_1 + \ell_2 + 1 - k, d.last$ )

```

O Código 4.3 faz exatamente o que foi discutido, e portanto devolve corretamente o k -ésimo elemento da versão d da deque, se k é válido, ou seja, está entre 1 e o número de elementos da deque d . A Tabela 4.1 mostra o consumo de tempo e espaço da implementação discutida nesse capítulo, usando as soluções de Ancestral de Nível dos Capítulos 1 e 2.

Operação	Representação binária	Skew-binary
<u>DEQUE</u> ()	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>PUSHFRONT</u> (q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>PUSHBACK</u> (q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>FRONT</u> (q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<u>BACK</u> (q)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<u>POPFRONT</u> (q)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(1)$
<u>POPBACK</u> (q)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(1)$
<u>K-TH</u> (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 duas implementações de Ancestral de Nível vistas na Parte I, onde n é o tamanho da estrutura da deque persistente.

Capítulo 5

Deque recursiva

Neste capítulo, apresentaremos uma segunda implementação persistente de deque. A implementação é discutida por Kaplan [7] e serve como base para a terceira implementação, apresentada no Capítulo 6.

5.1 Representação

A implementação usual de uma deque usando lista ligada utiliza uma lista duplamente ligada, e portanto para adicionar um elemento é necessário *alterar* campos de nós já existentes da lista, o que não resulta numa implementação funcional como ocorreu no caso da pilha. Uma solução é fazer como na implementação de deque do Capítulo 4, que utiliza uma arborescência para simular esta lista ligada. Neste capítulo, trataremos isto de outra forma.

Para conseguirmos manter a implementação funcional, mudaremos a representação de uma deque, usando um esquema recursivo. 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 as operações FRONT e BACK devolvem elementos desse conjunto T .

Uma deque persistente que armazena elementos do conjunto T é dada por um nó com três campos opcionais: *prefix*, *center*, e *suffix*; além de um campo obrigatório *size*. Um campo opcional pode não estar presente, e neste caso o campo assume o valor **null**. Os campos *prefix* e *suffix* armazenam valores de T , o campo *center* aponta para 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 está na deque, *suffix* representa um sufixo, e *center* os elementos do meio. A Figura 5.1 mostra uma possível deque persistente, que armazena a sequência (2, 3, 4, 5, 6, 7, 8). Note que os campos **null** (nós cortados) 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 da estrutura cada elemento *prefix* e *suffix* armazena 2^{i-1} elementos.

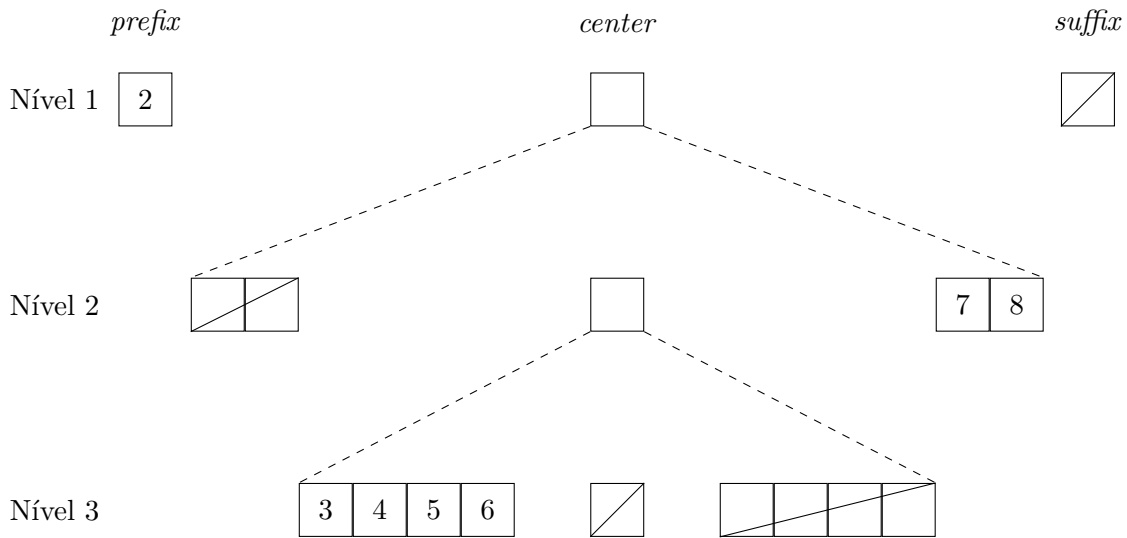


Figura 5.1: Representação de uma de/que persistente.

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, $\text{FRONT}(d)$ e $\text{BACK}(d)$ são chamadas apenas quando d não é vazia, e conseqüentemente 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 $(a, b).second = b$.

Código 5.1: Operações de acesso.

```

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

```

Proposição 5.1. A operação $\text{FRONT}(d)$ funciona corretamente, devolvendo o primeiro elemento de d .

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

Pela maneira como definimos a deque, se *prefix* não é **null** então ele é um prefixo de d , logo este é a resposta de $\text{FRONT}(d)$.

Se *prefix* é **null** e *center* também, o único elemento da deque é $d.\text{suffix}$ e o valor devolvido está correto. A base da indução, quando a deque tem um nível, sempre satisfaz um destes casos, logo é tratada corretamente.

Por fim, se *prefix* é **null** e *center* não é, como não armazenamos deque vazias, *center* tem algum elemento. O valor devolvido é então o primeiro elemento de $d.\text{center}$, e como esta é uma deque de pares ordenados de elementos de T , temos que o primeiro elemento do primeiro par de $d.\text{center}$ é o primeiro elemento de d . \square

Portanto, a função $\text{FRONT}(d)$ funciona corretamente. Note que $\text{BACK}(d)$ é simétrica a $\text{FRONT}(d)$. 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 não modificam a estrutura e consomem tempo $\mathcal{O}(\lg n)$, onde n é o número de elementos na estrutura da deque persistente.

5.3 Operações de modificação

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

Similarmente, para remover um elemento do início de d , se $d.\text{prefix}$ não for nulo, podemos remover esse elemento. Se ambos $d.\text{prefix}$ e $d.\text{center}$ forem nulos, podemos remover $d.\text{suffix}$. Caso contrário, podemos remover o elemento do início de $d.\text{center}$ e, já que os elementos de $d.\text{center}$ são pares, colocar apenas o segundo elemento em $d.\text{prefix}$, removendo assim o primeiro elemento da deque. Note que precisamos do elemento removido, então usaremos uma função auxiliar $\text{POPFRONTAUX}(d)$ 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. Assim, se apenas copiarmos todos os nós que iremos modificar, mantemos a estrutura funcional. Assumimos que $\text{new NODE}(pr, ct, sf, sz)$ 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 5.2. *A operação $\text{PUSHFRONT}(d, x)$ 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.

A base se dá quando d é nulo. Nesse caso a deque está vazia e a linha 3 devolve uma deque com x em seu campo *prefix*, o campos *size* com valor 1, e os outros campos nulos. Assuma então que d não é nulo e que a proposição vale para todas as deque com menos níveis que d .

Código 5.2: Operações de modificação para uma deque.

```

1: function PUSHFRONT( $d, x$ )
2:   if  $d = \text{null}$  :
3:     return new NODE( $x, \text{null}, \text{null}, 1$ )
4:   else if  $d.prefix = \text{null}$  :
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  $d.prefix \neq \text{null}$  and  $d.center = \text{null}$  and  $d.suffix = \text{null}$  :
12:     return ( $d.prefix, \underline{\text{DEQUE}}()$ )
13:   else if  $d.prefix \neq \text{null}$  :
14:     return ( $d.prefix, \text{new NODE}(\text{null}, d.center, d.suffix, d.size - 1)$ )
15:   else if  $d.center = \text{null}$  :
16:     return ( $d.suffix, \underline{\text{DEQUE}}()$ )
17:   else
18:     ( $x, c$ ) = POPFRONTAUX( $d.center$ )
19:     return ( $x.first, \text{new NODE}(x.second, c, d.suffix, d.size - 1)$ )

```

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

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

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 esta apenas devolve a deque devolvida por POPFRONTAUX(d). Consequentemente, POPBACK(d) está correta.

Proposição 5.3. *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 no **if** das linhas 11 e 12. Caso contrário, o **if** das linhas 13 e 14 devolve $d.prefix$ e cria um novo nó que tem **null** nesse campo, sem modificar d , portanto trata corretamente este caso.

Se $d.prefix$ é nulo e, se $d.center$ também é nulo, sabemos que d é uma deque com apenas um elemento, armazenado em $d.suffix$, pois assumimos que POPFRONTAUX(d) não é chamada se d é vazia. O **if** das linhas 15 e 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, quando $d.prefix$ é nulo mas $d.center$ não é, acionamos $POPFRONTAUX(d.center)$, que funciona corretamente 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 desse 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 assim ajustados. \square

As duas operações consomem tempo e espaço $\mathcal{O}(\lg n)$, onde n é o número de elementos da estrutura da deque persistente, 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 $\underline{K-TH}(d, k)$. Veja o Código 5.3.

Código 5.3: Implementação de $\underline{K-TH}(d, k)$.

```

1: function  $\underline{K-TH}(d, k)$ 
2:   if  $k = 1$  and  $d.prefix \neq \text{null}$  :
3:     return  $d.prefix$ 
4:   if  $k = d.size$  and  $d.suffix \neq \text{null}$  :
5:     return  $d.suffix$ 
6:   if  $d.prefix \neq \text{null}$  :
7:      $k = k - 1$ 
8:   if  $k$  is odd :
9:     return  $\underline{K-TH}(d.center, \lceil \frac{k}{2} \rceil).first$ 
10:  else
11:    return  $\underline{K-TH}(d.center, \lceil \frac{k}{2} \rceil).second$ 

```

Proposição 5.4. A função $\underline{K-TH}(d, k)$ devolve o k -ésimo elemento de d , para $k \leq d.size$.

Demonstração. As linhas 2 a 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 o **if** das linhas 6 e 7 corrige 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 $\underline{K-TH}(d.center, \lceil \frac{k}{2} \rceil)$ para encontrar o $\lceil \frac{k}{2} \rceil$ -ésimo par desta deque, pois o primeiro par guarda os elementos com índice 1 e 2, o segundo guarda os elementos com índice 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 $k + 1$, senão tem os elementos com índice $k - 1$ e k . \square

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

espaço da implementação discutida neste capítulo. Embora as operações de acesso FRONT, BACK e K-TH não indiquem consumo de espaço, já que não usam nenhum espaço *permanentemente*, estas ainda usam $\mathcal{O}(\lg n)$ de espaço temporariamente, devido a sua pilha de recursão.

Operação	Tempo/Espaço
<u>DEQUE</u> ()	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>PUSHFRONT</u> (q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
<u>PUSHBACK</u> (q, x)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
<u>FRONT</u> (q)	$\mathcal{O}(\lg n)$
<u>BACK</u> (q)	$\mathcal{O}(\lg n)$
<u>POPFRONT</u> (q)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
<u>POPBACK</u> (q)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
<u>K-TH</u> (q, k)	$\mathcal{O}(\lg n)$

Tabela 5.1: Consumo de tempo e espaço da implementação de deque deste capítulo, onde n é o número de elementos na estrutura da deque persistente.

Capítulo 6

Deque de Kaplan e Tarjan

Apresentaremos neste capítulo uma modificação da 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].

Assim como na deque recursiva do Capítulo 5, a estrutura da deque deste capítulo será composta por níveis, cada um armazenando um prefixo e um sufixo da deque, e um ponteiro para o próximo nível da deque, que armazena pares dos elementos desse nível. A diferença se dá pois, em vez de o prefixo e sufixo armazenarem zero ou um elemento, como na deque recursiva, nesta estrutura o prefixo e sufixo armazenam de zero a cinco elementos. Uma estratégia é usada para tentar manter cada uma destas extremidades balanceadas, de forma que seja possível executar as operações de forma mais rápida, já que em geral será possível adicionar ou remover um elemento alterando apenas o primeiro nível. Observe a Figura 6.2 e suas similaridades com a Figura 5.1.

Para apresentar esta estratégia, na Seção 6.1 observamos a similaridade das operações nessas estruturas com a operação de incremento em contadores binários e então, analisando um algoritmo mais eficiente de incremento em contadores binários, é possível utilizar a mesma ideia para tornar a implementação da deque mais eficiente.

6.1 Contadores binários

6.1.1 Deque persistente como um contador binário

Note que, na estrutura do Capítulo 5, se adicionarmos elementos apenas usando $\text{PUSHFRONT}(d, x)$, 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 à operação de incrementar 1 em um contador binário. Podemos considerar que cada nível da estrutura do Capítulo 5 é um dígito: 0 se *prefix* do nível é nulo ou 1 se *prefix* do nível é não nulo. O primeiro nível é o dígito menos significativo. Assim, adicionar um elemento usando $\text{PUSHFRONT}(d, x)$ é equivalente a incrementar de 1 o contador binário: se o dígito menos significativo é 0, esse dígito se torna 1; se ele é 1, então ele vira 0 e propagamos o aumento de 1 para o próximo dígito. Em $\text{PUSHFRONT}(d, x)$, se *prefix* é nulo, então *prefix* passa a apontar para x ; senão, ele passa a valer **null** e formamos um par com x e o elemento que era apontado por *prefix*, e incluímos esse par no próximo nível da estrutura.

Apesar de sabermos que incrementar 1 em um contador binário consome tempo amortizado $\mathcal{O}(1)$, sabemos que algumas vezes incrementar 1 pode levar tempo $\Theta(\lg n)$, onde n é o valor máximo que o contador atinge. Como estamos em uma estrutura persistente, podemos efetuar essa operação cara repetidas vezes na mesma versão, e não é possível manter o custo amortizado $\mathcal{O}(1)$.

6.1.2 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 b dígitos x_0, x_1, \dots, x_{b-1} , então o seu valor é $\sum_{i=0}^{b-1} x_i 2^i$. 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 que o contador é *regular*. Um contador com apenas 0s é claramente regular, e vamos mostrar como incrementar de 1 um contador binário regular mudando apenas $\mathcal{O}(1)$ dígitos e mantendo sua regularidade.

Suponha que temos um contador binário regular, e queremos incrementá-lo de 1. Pela regularidade, $x_0 < 2$, e portanto é possível incrementar esse dígito sem causar um estouro. No entanto, 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. Procuramos então o primeiro dígito 2 e, se um tal dígito existir, realizamos um procedimento FIX nele. Se não existir nenhum dígito 2, o contador é regular.

Caso exista, seja i o índice do primeiro dígito 2. Um procedimento FIX consiste em transformar esse dígito em 0 e incrementar de 1 o 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. O lado esquerdo representa como eram os dígitos antes do incremento e o lado direito, depois do incremento. 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.

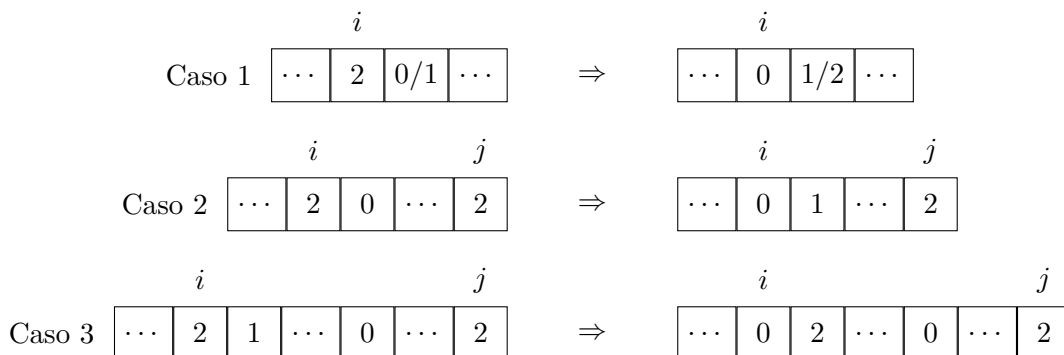


Figura 6.1: Casos possíveis para o procedimento FIX. O “...” representa zero ou mais dígitos 0 ou 1.

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, alterando no máximo três dígitos (x_0 , x_i e x_{i+1}). 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 esse incremento. Suponha 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.

Código 6.1: Incrementa de 1 o contador binário regular x .

```

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

```

A função $\text{ADD1}(x, p)$ no Código 6.1 incrementa de 1 o contador redundante regular x que tem os índices de seus dígitos 2 armazenados em p .

6.2 Visão geral

Como discutido na Subseção 6.1.1, a implementação de deque apresentada no Capítulo 5 tem uma certa relação com contadores binários. Iremos então usar a ideia de contadores binários redundantes, discutida na Subseção 6.1.2, 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 efêmeras que armazenam de zero a cinco elementos. Chamamos estas deques efêmeras de *sub-deques*. Na Figura 6.2, a sub-deque de sufixo do nível 2 tem os pares (8, 9) e (9, 10).

Cada nível será associado a um dígito 0, 1 ou 2, assim como em 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 implementação do Código 6.1. A deque da Figura 6.2 é associada ao contador com valor 112.

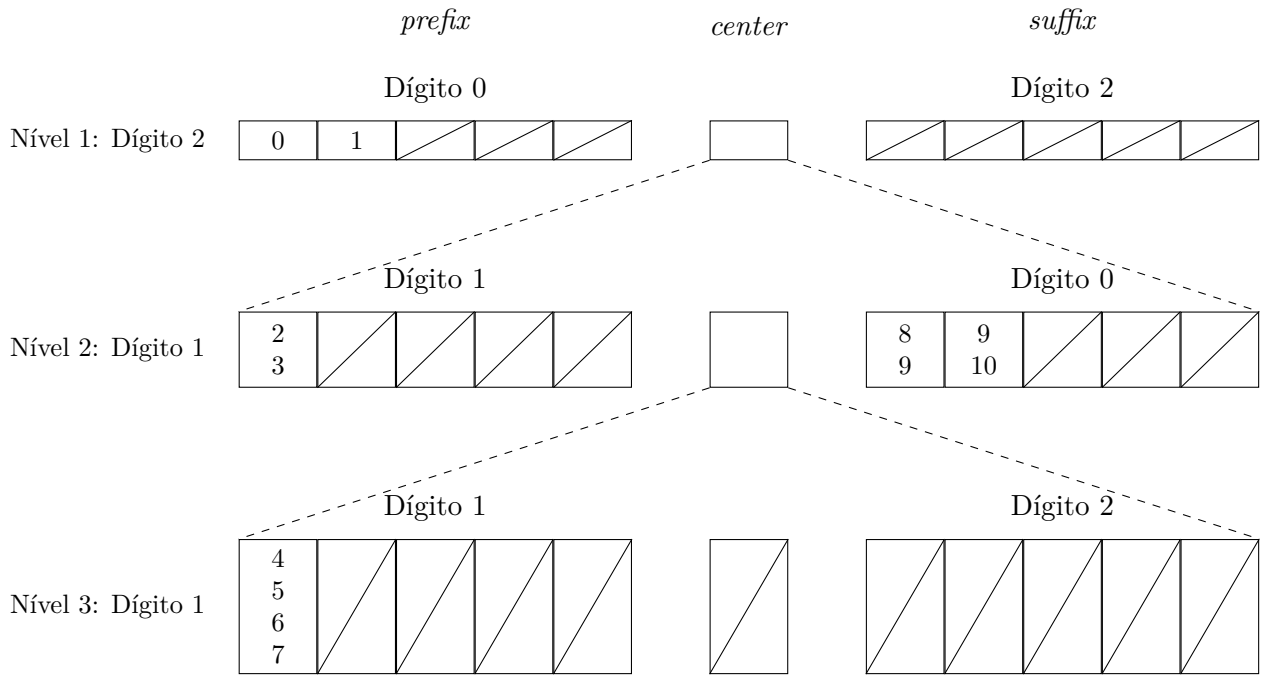


Figura 6.2: Deque persistente que armazena a sequência (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10).

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 dois ou três elementos, 1 se tem um ou quatro elementos, e 2 se tem zero ou cinco 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 zero ou cinco 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 dois 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 tanto de seu prefixo quanto de seu sufixo. Na Figura 6.2, o prefixo do nível 2 tem um elemento, logo tem dígito 1, já seu sufixo tem dois elementos, logo dígito 0. O dígito do nível 2 é então o máximo destes valores (ou seja, 1), indicando que tanto o prefixo quanto o sufixo do nível 2 podem adicionar e remover pelo menos um elemento.

Note que, na Figura 6.2, o dígito do nível 3 não é o máximo dos dígitos de suas sub-deques. Isto se dá pois este é o último nível, e desta forma sua sub-deque não vazia é tanto prefixo quanto sufixo deste nível. Neste caso específico, a quadrupla (4, 5, 6, 7) é tanto o primeiro quanto o último elemento do nível 3. Assim como na deque recursiva (linha 6 do Código 5.1), é necessário tratar de forma especial esse caso. Dizemos então que o dígito do último nível é o dígito de sua sub-deque não vazia (neste caso, 1). 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 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 têm dígito 0 ou 1, e então é possível adicionar ou remover um elemento de qualquer uma dessas sub-deques. Se estamos no caso degenerado, ou seja se a deque tem apenas um nível, 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, aumentar ou continuar igual). 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 eles e inserimos esse par 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 em questão. Dessa forma, ao final do FIX, cada uma das sub-deques do nível i tem dois ou três 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 do nível i para adicionar um par no *início* do prefixo do nível $i + 1$. O procedimento FIX foi descrito 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 partir desta seção, representaremos as deques de forma concisa, já que não nos importa mais especificamente quantos ou quais elementos estão em cada prefixo ou sufixo de cada nível, apenas o dígito de cada nível. A Figura 6.3 mostra a deque da Figura 6.2 em versão concisa, e o restante das representações de deque neste capítulo serão desta forma.

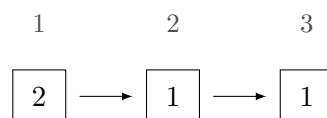


Figura 6.3: Representação concisa da deque da Figura 6.2: os números sobre as caixas indicam o nível e os números dentro das caixas indicam o dígito do nível. Omitem-se os elementos nessa representação.

A primeira dificuldade é encontrar (e modificar) o primeiro nível com dígito 2, pois este nível pode estar arbitrariamente fundo na lista de níveis. 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, além da deque persistente, 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. Isso, porém, não funciona.

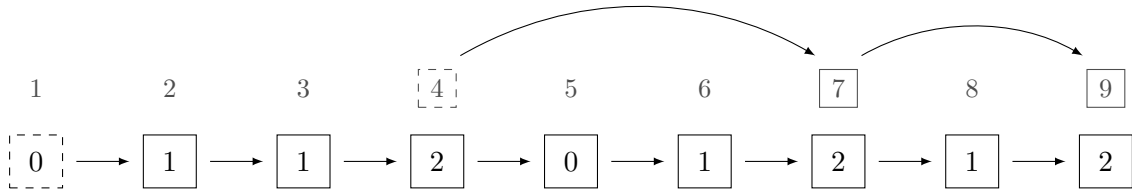


Figura 6.4: Primeira ideia de representação da deque persistente.

Considere a Figura 6.4. Os nós tracejados são os nós de entrada, ou seja, os nós que armazenamos nesta versão da deque; no caso, 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 esse dígito em 0 e o dígito do nível 5 em 1.

Se copiarmos apenas os nós modificados, a estrutura fica como na Figura 6.5, 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. Como não modificamos nenhum vértice, esta estrutura é funcional (e persistente), porém ela está errada, já que a nova versão não representa a deque corretamente.

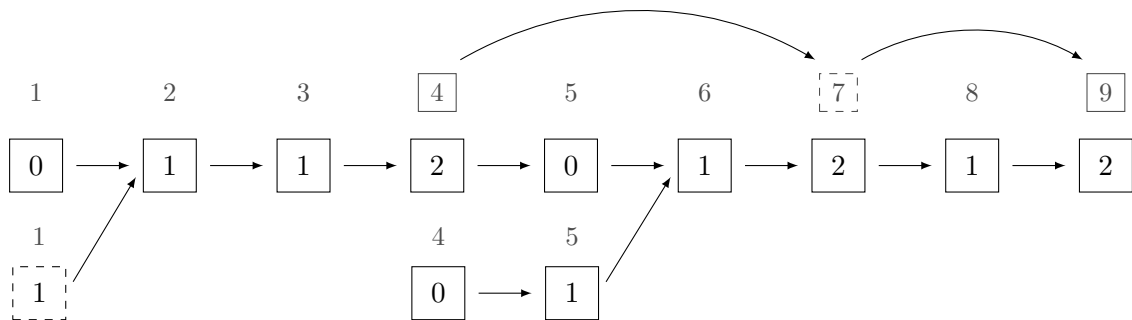


Figura 6.5: Deque inválida gerada depois de uma operação de modificação na deque da Figura 6.4, seguida de um FIX.

Para implementar corretamente 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 implementação da deque persistentes do Capítulo 5, isso é sempre verdade pois copiamos um prefixo dos níveis, e cada nível apenas aponta para o próximo nível, e nas implementações de pilha, fila e deque dos Capítulos 3 e 4, isto também vale.

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 têm dígito 1. Faz-se então uma pilha em que cada elemento é o topo de uma destas sub-pilhas. Desse modo, o procedimento FIX é sempre realizado no nível 1 ou no primeiro elemento da segunda pilha, já que ou um deles é o primeiro dígito 2 ou a estrutura já é regular.

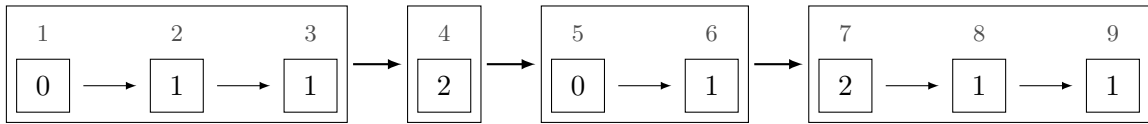


Figura 6.6: Segunda ideia de representação da deque persistente.

A Figura 6.6 mostra como seriam essas pilhas na mesma deque da Figura 6.4. Faremos a implementação destas pilhas adicionando um campo a cada nó, o campo *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**.

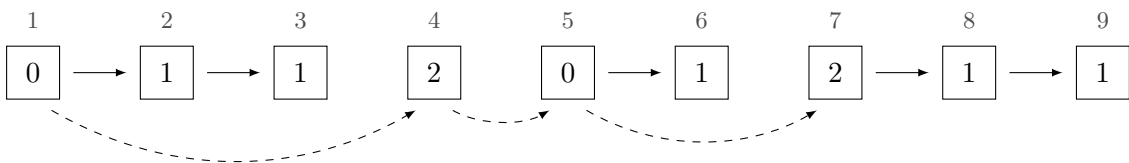


Figura 6.7: Mais detalhes da segunda ideia de representação da deque persistente.

A Figura 6.7 mostra como fica a estrutura, baseando-se nessa implementação, onde as flechas tracejadas indicam os ponteiros *next* não nulos. Removemos alguns ponteiros armazenados nos campos *child*, mais precisamente os que apontam para a próxima sub-pilha, isso é feito pois a próxima sub-pilha deve ser acessada pelo ponteiro *next* do primeiro elemento da sub-pilha. Dessa forma, não é mais possível acessar nós de versões passadas. Com isso, a navegação pelos níveis fica um pouco mais complexa, já que um nó pode ter o campo *child* nulo e ainda assim existir um nó no próximo nível. A Figura 6.8 mostra a estrutura após as mesmas modificações que foram aplicadas na Figura 6.5. Note que agora cada deque pode ser armazenada apenas com um ponteiro para o seu primeiro nível, já que não é mais necessário armazenar o ponteiro para a pilha separadamente.

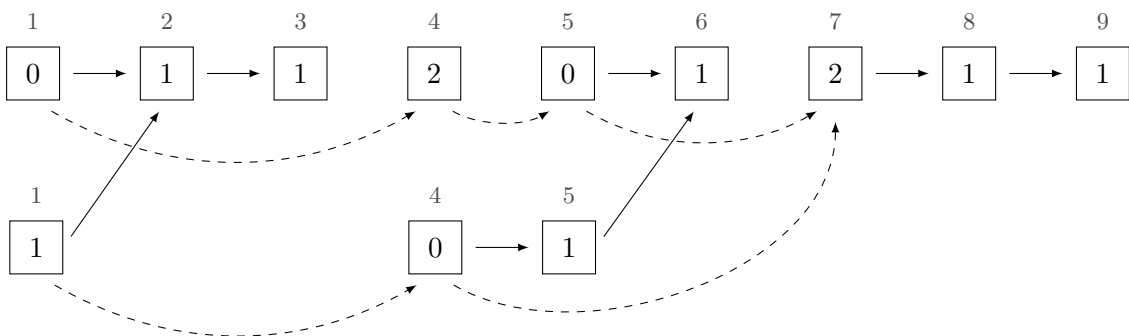


Figura 6.8: Deque da Figura 6.7 após uma operação de modificação seguida de um FIX.

6.5 Representação

A representação de um nível da deque será similar à do Capítulo 5. Um nível da deque é representado por um nó u , que terá os campos *prefix*, *suffix*, *child* e *next*. Os campos *prefix* e *suffix*, como anteriormente, guardam um prefixo e um sufixo da deque representada por u , mas agora estes campos são deques efêmeras de tamanho até cinco. Chamamos estas deques efêmeras de cada nível de *sub-deques*. Consideramos que $\{\}$ cria uma deque efêmera 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 a função SIZE() devolve o número de elementos na deque efêmera. Por exemplo, se d é uma deque efêmera, $d.POPBACK()$ remove o último elemento de d e o devolve, modificando d .

Como discutido na seção anterior, o campo *next* de u aponta para o próximo nível com dígito diferente de 1, se tal nível existir e se u não tiver dígito 1, ou se u for o primeiro nível. O campo *child* do nó u aponta para o nó do próximo nível, se tal nível existir e tiver 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 semirregularidade 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 auxiliar $FIXDEQUES(l, r, L, R)$. Essa função recebe duas sub-deques efêmeras l e r , o prefixo e sufixo do nível i da deque que estamos realizando o procedimento FIX e deixa ambas com dois ou três elementos, possivelmente modificando as sub-deques L e R , o 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 três elementos de nível i entre todas as deques da chamada, isto é, se $SIZE(l) + SIZE(r) + 2SIZE(L) + 2SIZE(R) > 3$.

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

Caso 1. Existem no máximo três elementos de nível i entre os níveis i e $i + 1$.

Adicione todos os elementos dos níveis i e $i + 1$ no prefixo do nível i , mantendo a ordem, e remova o nível $i + 1$.

Caso 2. Existem mais do que três elementos de nível i entre os níveis i e $i + 1$.

Sejam a e b os nós que representam os níveis i e $i + 1$, respectivamente; execute a função $FIXDEQUES(a.prefix, a.suffix, b.prefix, b.suffix)$. 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 um 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, dois ou três elementos. De fato, é impossível que exista apenas um elemento de nível i entre os níveis i e $i + 1$, pois isso implicaria que

não existem elementos em $i+1$, e o nível $i+1$ teria dígito 2 (uma contradição pela semirregularidade da estrutura), ou que o nível $i+1$ não existe, e então o nível i é o último e tem exatamente um 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 na descrição da 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 de dígito 2, e L e R são o prefixo e sufixo do nível $i+1$ de dígito 0 ou 1. É possível que o nível $i+1$ não exista na deque atual; nesse caso L e R são deques efêmeras vazias.

Código 6.2: Função `FIXDEQUES`

Require: $l.SIZE() + r.SIZE() + 2 \cdot L.SIZE() + 2 \cdot R.SIZE() > 3$

```

1: function FIXDEQUES( $l, r, L, R$ )
2:   if  $l.SIZE() \geq 4$  :
3:      $b = l.POPBACK()$ 
4:      $a = l.POPBACK()$ 
5:      $L.PUSHFRONT((a, b))$ 
6:   if  $r.SIZE() \geq 4$  :
7:      $a = r.POPFRONT()$ 
8:      $b = r.POPFRONT()$ 
9:      $R.PUSHBACK((a, b))$ 
10:  if  $l.SIZE() \leq 1$  :
11:    if  $L.SIZE() \neq 0$  :
12:       $(a, b) = L.POPFRONT()$ 
13:    else
14:       $(a, b) = R.POPFRONT()$ 
15:     $l.PUSHBACK(a)$ 
16:     $l.PUSHBACK(b)$ 
17:  if  $r.SIZE() \leq 1$  :
18:    if  $R.SIZE() \neq 0$  :
19:       $(a, b) = R.POPBACK()$ 
20:    else
21:       $(a, b) = L.POPBACK()$ 
22:     $r.PUSHFRONT(b)$ 
23:     $r.PUSHFRONT(a)$ 

```

A função, para cada sub-deque, se esta tem pelo menos quatro elementos, remove dois destes e os insere na sub-deque correspondente do nível $i+1$. Se a sub-deque tem no máximo um elemento, então um par é removido da sub-deque correspondente do nível $i+1$ e os dois elementos do par são adicionados na sub-deque. Dessa forma, ao final do procedimento, ambas as sub-deques têm dois ou três 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 `ifs` das linhas 2 e 6 do Código 6.2 tratam os casos de remover elementos do nível i para adicioná-los ao nível $i+1$, e os `ifs` das linhas 10 e 17 tratam os casos de remover um par do

nível $i + 1$ e adicionar seus elementos ao nível i . Se alguma das sub-deques do nível $i + 1$ está 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. Assim os **ifs** das linhas 11 e 18 tratam corretamente esse caso. Dessa forma, o código faz como descrevemos, resta provar que as operações nas sub-deques L e R são sempre válidas, ou seja, nunca é executada uma operação de PUSH em uma sub-deque cheia ou uma operação de POP quando o nível $i + 1$ está vazio.

As deques L e R não têm cinco 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 **ifs** das linhas 2 e 6 sempre funcionam. Além disso, no máximo um entre os **ifs** das linhas 2 e 10 é executado, assim como entre os **ifs** das linhas 6 e 17. Dessa forma, se algum dos primeiros dois **ifs** 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 **ifs** 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 pode ser inválida são:

- Os últimos dois **ifs** são executados, ou seja, se l e r têm 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 três elementos de nível i entre todas as sub-deques, o único caso que isso pode acontecer é quando as deques l e r têm exatamente um elemento, e existe exatamente um par em uma das deques L ou R . Mas nesse caso o dígito do nível i seria 1, e FIXDEQUES não seria chamada.
- Apenas um dos últimos dois **ifs** é executado, e o nível $i + 1$ está vazio, ou seja, o nível i é o último. Isso ocorre se uma entre l e r tem dois ou três elementos, e a outra tem no máximo um. Mas este caso não ocorre pois o dígito do nível i seria 1, e FIXDEQUES não seria chamada.

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 $\text{FIX}(a)$, uma função que recebe um nó de nível i com dígito 2 e faz alterações nele para transformar 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. Assim não modificamos o nó original de nível $i + 1$. Para não modificar um nó da estrutura, quando usarmos esta função na implementação das operações da deque em seções futuras, chamaremos a função com uma cópia do nó de nível i . Usamos a função $\text{COPY}(x)$ para devolver uma cópia de um nó x , em tempo constante. Se x for **null**, essa função devolve um nó vazio (com as sub-deques vazias e os campos nulos).

A função $\text{DIGIT}(a, last)$ devolve o dígito do nó a , onde $last$ é **true** se não existe nível depois de a , e **false** caso contrário. O parâmetro $last$ 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 6.7, onde 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 10-18 copiam o nó de nível $i + 1$ na variável b , criando um nó vazio se este não existir. Se o nó de nível $i + 1$ existir, ele pode estar tanto no campo *child*, se tiver dígito 1, quanto

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

```

no campo *next*, se tiver dígito 0. Também é armazenado na variável *last* o valor **true** se não existe nível $i + 2$, e **false** caso contrário. A notação [predicado] de Iverson 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 um elemento.

O Caso 2 é tratado nas linhas 31-34. Na Seção 6.7 argumentamos que a função `FIXDEQUES` está correta. 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 o ponteiro *next* de a 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 topo de uma sub-pilha, já que tem dígito 0.

6.9 Implementação das operações

Com a função `FIX` implementada, é fácil implementar as operações da deque com consumo de tempo constante. Tendo em vista a 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.

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 **ifs** 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 um pouco mais complexa porém similar ao Código 5.3. A Tabela 6.1 compara o consumo de tempo e espaço de

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:     |   a.prefix.POPFRONT()
16:   else
17:     |   a.suffix.POPFRONT()
18:   CHECK(a)
19:   return a
20: function FRONT(a)
21:   if a.prefix.SIZE() ≠ 0 :
22:     |   return a.prefix.FRONT()
23:   else
24:     |   return a.suffix.FRONT()

```

todas as dequeues persistentes apresentadas neste trabalho. Note que a operação K-TH da deque de Kaplan e Tarjan consome temporariamente espaço $\mathcal{O}(\lg n)$ devido a sua pilha de recursão.

Operação	Representação binária	Skew-binary	Recursiva	Kaplan e Tarjan
<u>DEQUE</u> ()	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>PUSHFRONT</u> (<i>q</i> , <i>x</i>)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>PUSHBACK</u> (<i>q</i> , <i>x</i>)	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>FRONT</u> (<i>q</i>)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(1)$
<u>BACK</u> (<i>q</i>)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(1)$
<u>POPFRONT</u> (<i>q</i>)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>POPBACK</u> (<i>q</i>)	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$	$\mathcal{O}(1)/\mathcal{O}(1)$
<u>K-TH</u> (<i>q</i> , <i>k</i>)	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 6.1: Comparação do consumo de tempo e espaço das implementações das dequeues descritas nos Capítulos 4, 5 e 6, onde n é o tamanho da estrutura da deque persistente.

Capítulo 7

Técnicas gerais

Nos capítulos anteriores, discutimos como tornar persistentes estruturas de dados específicas (pilhas, filas e deque). 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*, 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**, que indica que não estão apontando para nenhum nó. 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 deque 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 $\lceil \lg d \rceil$, 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.
- Altera-se um nó de entrada. Seu novo valor deve ser **null** ou um nó do conjunto de acessos.

O tempo consumido por uma 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 nó ali, modificando o ponteiro do nó do conjunto de acessos que deve apontar para ele. Logo esta operação consome tempo $\mathcal{O}(h)$ e tempo de modificação $\mathcal{O}(1)$.

Essas definições descrevem uma estrutura de dados efêmera. 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 $i \rightarrow j$ 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 $i \rightarrow k$), a operação de modificação que criou a

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 3.1 da página 20.

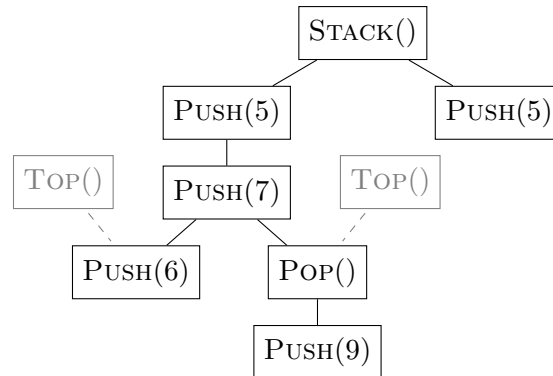


Figura 7.1: Árvore de versões do Exemplo 3.1, com argumentos para fazer cada operação de modificação indicados. Os nós mais claros (cinzentos) indicam as operações de acesso, conectadas à versão na qual são realizadas.

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

1. Ao acessar um vértice v_i na DFS, a estrutura está na versão j , onde $j \rightarrow i$.
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 recursivamente em todas as versões descendentes de i .
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 processar um a um os nós que são filhos de v_i recursivamente, já que, devido ao passo 4, quando o acesso a cada um deles termina, a estrutura volta a estar na versão i . No passo 4 de v_i , revertemos as mudanças feitas no passo 2 para v_i .

Para reverter mudanças de forma genérica, sempre que um passo de modificação altera um campo de um nó, guardamos uma tripla $(u, field, oldValue)$ indicando que o campo $field$ do nó u 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 nó para armazenar os nós de entrada.

Como não impusemos restrições nas operações de mudança, elas podem modificar um mesmo campo muitas vezes, ou utilizar o valor de um campo previamente modificado. Logo, a ordem em que as operações ocorreram é relevante e, em particular, 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

Código 7.1: Persistência total offline

```

1: function DFS( $v_i$ )
2:    $mod = \text{STACK}()$ 
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  $v_k \in$  lista de adjacência de  $v_i$  :
6:     DFS( $v_k$ )
7:   while  $mod.\text{SIZE}() > 0$  :
8:      $(u, field, oldValue) = mod.\text{POP}()$ 
9:      $u.field = oldValue$ 
10: function TOTALPERSISTENCEOFFLINE()
11:   Montar a árvore de versões, com raiz  $v_0$ .
12:   DFS( $v_0$ )

```

na linha 4 deve iterar pela lista de operações de acesso do vértice v_i e aplicá-las. Como assumimos 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 $\text{DFS}(v_i)$ é chamada com a versão j da estrutura de dados, onde $j \rightarrow i$, 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 aciona, para cada filho de v_i , a função DFS neste recursivamente. 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 $i \rightarrow x$, temos que a função aplica corretamente as operações de acesso das versões na subárvore de v_x , e ao final da chamada de $\text{DFS}(v_x)$, a estrutura está novamente na versão 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 restaura a estrutura para a versão j , completando a prova. \square

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 o Capítulo 6 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 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 após a sua criação. Quando a estrutura não funciona dessa forma, podemos tentar impor 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 os 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.

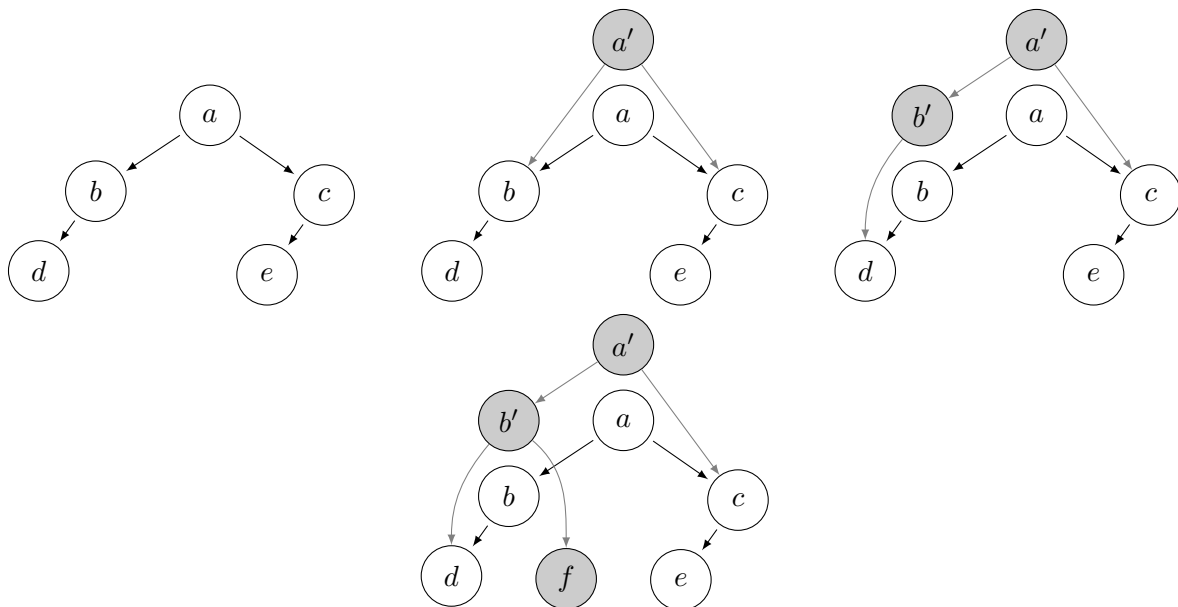


Figura 7.2: Adição de um nó f à direita de b com uma implementação totalmente funcional. Em cinza, os nós criados ou copiados durante a adição.

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 do nó 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 os 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 de uma estrutura ligada em particular formam uma floresta direcionada, onde os nós de entrada são as raízes e as arestas vão em direção contrária às raízes, para cada nó u existe exatamente um caminho de um nó de acesso até u , logo se u é adicionado ao conjunto de acesso, temos a garantia que todos os nós que apontam pra ele já estão no conjunto de acesso, não importando a ordem na qual foram feitas as adições ao conjunto de acessos. Dessa forma, basta copiar os nós visitados durante operações de modificação e a estrutura em questão se torna funcional e persistente. 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.

Esse método torna a estrutura totalmente persistente com aumento no consumo de tempo de $\mathcal{O}(M + A)$, levando em conta o tempo de copiar os nós visitados em operações de modificação e o tempo de modificar os ponteiros que apontam para tais nós, 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 um nó acessado em uma operação de modificação mesmo que nenhum campo deste tenha sido alterado.

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 et al. [5].

Modifica-se cada nó da estrutura para que cada campo seja substituído por um vetor que armazena pares (i, v) , indicando que a i -ésima operação de modificação alterou o valor daquele campo para v . Em vez de modificar um campo de um nó, adiciona-se um par no vetor correspondente, indicando a versão da estrutura que está sendo criada 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 pelos 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 está armazenada em ordem. Diferente das outras técnicas, esta aumenta também 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 garante 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 fica 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 limitado por uma constante. Especificamente, 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 têm 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 está inicialmente vazio, e armazena mudanças feitas ao nó depois de sua criação.

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

Se algum dos nós que apontam para u não tiver espaço livre em seu vetor *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 tal cópia ainda 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, dizemos que cada nó u *representa* o nó correspondente da estrutura efêmera no intervalo de versões $[u.\mathcal{T}, u.\text{copy}.\mathcal{T} - 1]$ se este tiver cópia. Caso contrário, o nó *representa* as versões desde $u.\mathcal{T}$ até a versão atual do nó correspondente na estrutura efêmera, e é chamado de *nó ativo*. Como estamos tratando de persistência parcial, apenas nós ativos são modificados.

7.5.1 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 7.5.2, 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 7.5.3, 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.

7.5.2 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 $version$, ou seja, $version \geq u.T$ e $version < u.copy.T$, se $u.copy$ é não nulo.

```

1: function ACCESS( $u, field, version$ )
2:    $value = u.field$ 
3:   for ( $version', field', value'$ )  $\in u.changes$  :
4:     if  $field = field'$  and  $version' \leq version$  :
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 $version$. Para realizar uma operação de acesso na versão i , basta acessar os nós de entrada da versão i , utilizando o vetor $entry$, e realizar os acessos usando a função de interface ACCESS. O aumento de complexidade de tempo é constante por passo de acesso.

7.5.3 Modificação

Em uma operação de modificação, alterar um nó do conjunto de acessos pode levar à cópia de outros nós que já estão no conjunto de acessos. Vamos garantir que no máximo uma cópia de cada nó seja feita por operação de modificação. 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  $u = \text{null}$  or  $u.copy = \text{null}$  :
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 **nulls**. 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 $\text{CHANGEPARENT}(u, a, b)$ modifica o vetor $u.parents$, trocando uma ocorrência de a por b . A função $\text{CHANGEPOINTER}(u, field, value)$ 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  $u.field \neq \text{null}$  :
8:     CHANGEPARENT( $u.field, u, \text{null}$ )
9:      $u.field = \text{ACTIVE}(value)$ 
10:  if  $u.field \neq \text{null}$  :
11:   CHANGEPARENT( $u.field, \text{null}, u$ )

```

▷ $value$ pode já ter sido copiado.

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

A função $\text{MODIFY}(u, field, value)$ altera 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:

Código 7.5: Alteração 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:      $u.field = value$ 

```

- 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 alteram 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:    $u'.changes = \{\}$                                   ▷ Inicializa  $changes$  com vetor vazio.
4:    $u'.\mathcal{T} = current$ 
5:   for ( $version', field', value' \in u.changes$  :
6:      $u'.field' = value'$ 
7:   if  $u$  é nó de entrada :
8:     Trocar  $u$  por  $u'$  na posição correta do vetor  $entry$ .
9:   for  $pointer \in$  campos de ponteiros :
10:    if  $u'.pointer \neq null$  :
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  $v \neq null$  :
16:        $field =$  campo de  $v$  tal que  $ACCESS(v, field) = u$ .
17:       if  $v.\mathcal{T} = current$  :
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 $\text{RAWCOPY}(u)$ cria 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 $u.\text{changes}$ 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 $\text{current} - 1$, 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 $\text{ACCESS}(v, \text{field}) = u$, queremos que $\text{ACCESS}(v, \text{field}) = u'$. 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 $v.\text{changes}$ — Adicionamos a modificação ao vetor changes , ou seja, adicionamos a tripla $(\text{current}, \text{field}, u')$ a changes .
3. v não tem espaço livre em $v.\text{changes}$ — 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 $v.\text{copy}$).

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

7.5.4 Análise de espaço e tempo

A ideia do método é copiar os nós quando fazemos alterações em passos de modificação, mas registram-se algumas alterações no próprio 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 escolhermos 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

$$\Phi(E_i) = \sum_{u \text{ é ativo}} |u.\text{changes}|,$$

ou seja, o número de alterações armazenadas em todos os vetores *changes* de todos os nós ativos em E_i . É fácil ver que $\Phi(E_i)$ nunca é negativo, e claramente $\Phi(E_0) = 0$.

Segundo a definição na Seção 7.1, um passo de modificação pode criar um novo nó, alterar um nó de entrada, ou, alterar um campo de um nó existente, usando para isso a função de interface MODIFY. Vamos analisar cada um destes casos.

Se um passo de modificação cria um novo nó, o potencial não se modifica, já que o novo nó tem *changes* vazio. Seja N_i o número de nós criados de tal forma durante a i -ésima operação de modificação. É óbvio que $O_i \geq N_i$.

Se um nó de entrada é trocado, o potencial não se altera, e nenhum nó é criado ou copiado.

Por fim, ao alterar um campo de um nó existente, usando a função de interface MODIFY, nós podem ser copiados pelo uso da função COPY. Vamos considerar então nós criados pela função COPY durante a i -ésima operação de modificação. Esta é uma função auxiliar e não é chamada diretamente durante os passos de modificação, apenas por MODIFY (linha 4 do Código 7.5) e recursivamente pela própria COPY (linha 22 do Código 7.6). Seja C_i o número de chamadas de COPY feitas por MODIFY e D_i o número de chamadas de COPY feitas por COPY. Vale que $O_i = N_i + C_i + D_i$.

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 $\text{COPY}(u)$ é chamada recursivamente, isso ocorreu pois u tinha o vetor *changes* cheio, logo o potencial diminui em *in* nesse caso, já que u deixa de ser ativo e um novo nó ativo é criado com *changes* vazio.

O **for** da linha 13 do Código 7.6 pode adicionar mudanças aos vetores *changes* dos nós que apontam para u (linha 20), ou seja, aumentar o potencial. 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, já que a linha 20 não pode ser executada na mesma iteração em que a linha 22 é executada. Logo $y \leq (C_i + D_i) \cdot in - D_i$.

Dessa forma, o potencial só se altera por causa de chamadas recursivas de COPY, ou por causa de adições ao vetor *changes* de algum nó ativo, ou seja, vale que $\Phi(E_i) = \Phi(E_{i-1}) - D_i \cdot in + y$. Utilizando a notação definida acima temos:

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

A igualdade (1) vale pois $O_i = N_i + C_i + D_i$ e $\Phi(E_i) = \Phi(E_{i-1}) - D_i \cdot in + y$ 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 complexos 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 (ignorando as mudanças de cores) 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 Sedgwick e Wayne [14], 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 o consumo de $\mathcal{O}(m \lg m)$ de espaço é aceitável, é possível fazer uma implementação funcional de árvore rubro-negra. Esta implementação consome mais espaço que a implementação que vamos apresentar neste capítulo, mas é mais simples 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**.

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.

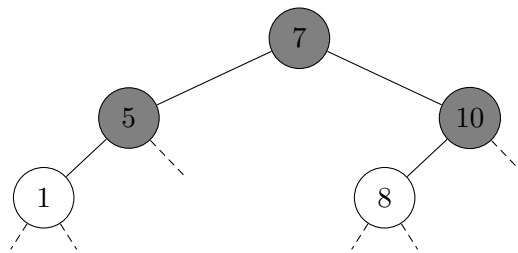


Figura 8.1: Exemplo de árvore rubro-negra. Como nas outras figuras do capítulo, os nós negros têm fundo cinza e os nós vermelhos têm fundo branco. Nas próximas figuras, o valor armazenado em cada nó não é especificado.

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 $3\lceil \lg(n) \rceil$, 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, 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*, *extraT* — 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 *extraT* = -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).

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.extraT ≠ -1 and u.extraSide = side and version ≥ u.extraT :
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.

```

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 *extraT* 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, no caso, a raiz da árvore, para cada versão. Note que este vetor é equivalente ao vetor `entry` discutido na Seção 7.5.

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  $u$ 

```

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.

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, assumimos que apenas ponteiros são modificados em nós de uma árvore rubro-negra, nunca o valor dos nós.

Para diminuir o código, usamos o fato que cada nó tem apenas um campo extra e um ponteiro de volta. 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 $[CHILD(v, 1) = u]$ 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 que armazenam 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

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

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

```

1: function MODIFY( $u, side, v$ )
2:    $u = \text{ACTIVE}(u)$ 
3:   if  $u.\mathcal{T} < \text{current}$  :
4:      $u.\text{copy} = \text{COPY}(u)$ 
5:      $u = u.\text{copy}$ 
6:   if  $u.\text{child}[side] \neq \text{null}$  :
7:      $u.\text{child}[side].\text{parent} = \text{null}$ 
8:    $u.\text{child}[side] = \text{ACTIVE}(v)$ 
9:   if  $u.\text{child}[side] \neq \text{null}$  :
10:     $u.\text{child}[side].\text{parent} = u$ 
11: function COPY( $u$ )
12:    $u' = \text{RAWCOPY}(u)$ 
13:    $u'.\mathcal{T} = \text{current}$ 
14:   if  $u.\text{extra}\mathcal{T} \neq -1$  :
15:      $u'.\text{child}[u.\text{extraSide}] = u.\text{extra}$ 
16:      $u'.\text{extra}\mathcal{T} = -1$ 
17:   if  $\text{roots}[\text{current}] = u$  :
18:      $\text{roots}[\text{current}] = u'$ 
19:   for  $side \in \{0, 1\}$  :
20:     if  $u'.\text{child}[side] \neq \text{null}$  :
21:        $u'.\text{child}[side].\text{parent} = u'$ 
22:    $u.\text{parent} = \text{null}$ 
23:   if  $u'.\text{parent} \neq \text{null}$  :
24:      $v = u'.\text{parent}$ 
25:      $side = [\text{CHILD}(v, 1) = u]$ 
26:     if  $v.\mathcal{T} = \text{current}$  :
27:        $v.\text{child}[side] = u'$ 
28:     else if  $v.\text{extra}\mathcal{T} = -1$  :
29:        $v.\text{extra}\mathcal{T} = \text{current}$ 
30:        $v.\text{extra} = u'$ 
31:        $v.\text{extraSide} = side$ 
32:     else
33:        $v.\text{copy} = \text{COPY}(v)$ 
34:        $v.\text{copy}.\text{child}[side] = u'$ 
35:        $u'.\text{parent} = v.\text{copy}$ 
36:   return  $u'$ 

```

▷ Faz v filho de u do lado $side$.

▷ Limpando o campo extra.

que os nós na subárvore esquerda têm valores menores ou iguais e os nós da subárvore direita têm 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) Nenhuma propriedade é violada 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.

8.6.1 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

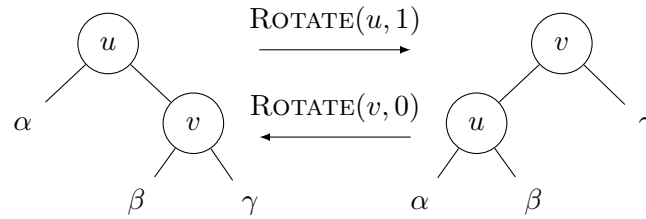


Figura 8.2: Exemplo de rotação direita e esquerda em uma ABB. Os símbolos α , β e γ indicam subárvores, possivelmente vazias.

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 $\text{ROTATE}(u, 1)$.

8.6.2 Subindo a violação

Se a árvore ainda tem violações no começo da iteração do laço, seja y o pai de x . Sabemos que x viola a propriedade 2, logo y é vermelho e não é a raiz (pois a propriedade 1 não é violada), 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 $x' = z$.



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

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.

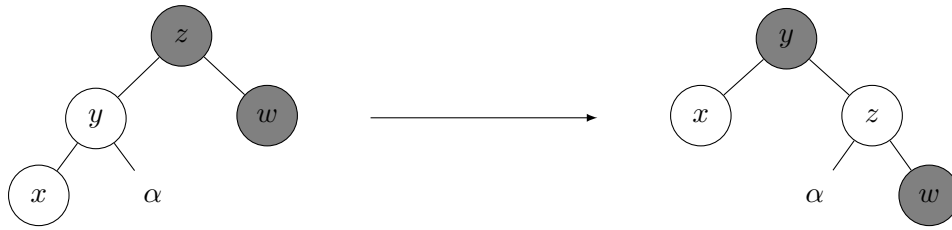


Figura 8.4: Aplicação do caso 2, assumindo que x é filho esquerdo. Os nós x e w (se existir) podem ter filhos.

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).

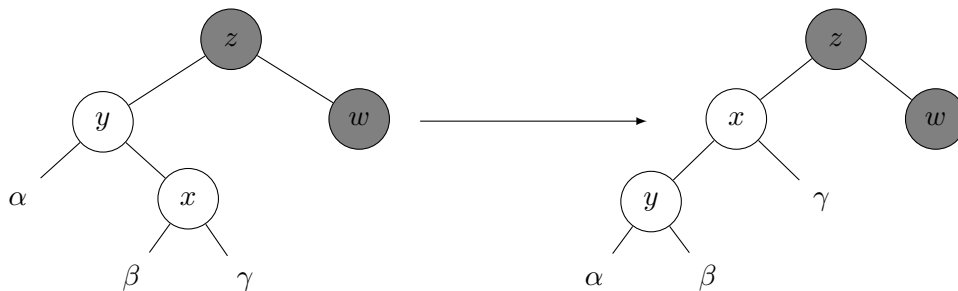


Figura 8.5: Aplicação do caso 3, assumindo que x é filho direito. O nó w , se existir, pode ter filhos.

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 a árvore é balanceada.

8.6.3 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.

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 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.

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)
14:    |     ▷ Arrumando a possível violação causada pelo nó x.
15:    |     while x.red and x.parent ≠ null and x.parent.red :
16:    |     |   y = x.parent
17:    |     |   z = y.parent
18:    |     |   sideX = [CHILD(y, 1) = x]
19:    |     |   sideY = [CHILD(z, 1) = y]
20:    |     |   w = CHILD(z, not sideY)
21:    |     |   if w ≠ null and w.red :           ▷ Caso 1.
22:    |     |     |   z.red = true
23:    |     |     |   y.red = false
24:    |     |     |   w.red = false
25:    |     |     |   x = z
26:    |     |   else
27:    |     |     |   if sideX ≠ sideY :           ▷ Caso 3.
28:    |     |     |     |   ROTATE(y, sideX)
29:    |     |     |     |   x, y = y, x           ▷ Trocando x e y.
30:    |     |     |     |   ROTATE(z, sideY)           ▷ Caso 2.
31:    |     |     |     |   ACTIVE(z).red = true
32:    |     |     |     |   ACTIVE(y).red = false
33:    |     |     |     |   break
34:    |     |     |   roots[current].red = false           ▷ Caso a raiz tenha sido pintada de vermelha.

```

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 laços 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, garantir 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:    $v = \text{CHILD}(u, \textit{side})$ 
3:    $\beta = \text{CHILD}(v, \textbf{not } \textit{side})$ 
4:   MODIFY( $v$ , not side, null)
5:   MODIFY( $u$ , side,  $\beta$ )
6:    $u = \text{ACTIVE}(u)$ 
7:   if  $u.\textit{parent} \neq \textbf{null}$  :
8:     | MODIFY( $u.\textit{parent}$ , [CHILD( $u.\textit{parent}$ , 1) =  $u$ ],  $v$ )
9:   else
10:  |  $\textit{roots}[\textit{current}] = \text{ACTIVE}(v)$  ▷ 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  $u.child[0] \neq \mathbf{null}$  :
3:      $u = u.child[0]$ 
4:   return  $u$ 
5: function TRANSPLANT( $u, x$ )
6:    $v = u.parent$ 
7:   if  $v \neq \mathbf{null}$  :
8:      $v.child[[v.child[1] = u]] = x$ 
9:   else
10:     $root = x$ 
11:   if  $x \neq \mathbf{null}$  :
12:     $x.parent = v$ 

```

Require: A árvore tem um nó com valor *value*.

```

13: function REMOVE( $value$ )
14:    $u = \text{FIND}(value)$ 
15:   if  $u.child[1] = \mathbf{null}$  :
16:     TRANSPLANT( $u, u.child[0]$ )
17:   else
18:      $x = \text{MINELEMENT}(u.child[1])$ 
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 .

Na operação `REMOVE($value$)`, 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 .

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.

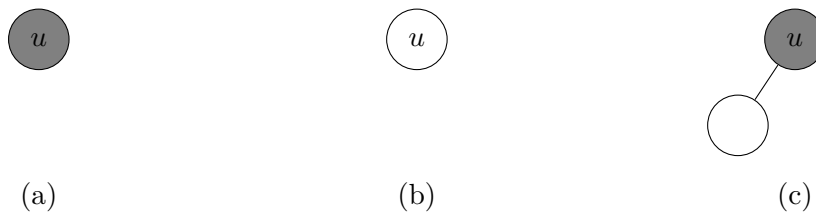


Figura 8.6: Possíveis subárvores rubro-negras com raiz u que não tem filho direito.

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 $\text{TRANSPLANT}(u, x)$ 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. Se u for raiz, a árvore final é vazia e segue todas as propriedades. Usamos então a função auxiliar ADDBLACK , que corrige 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 $\text{TRANSPLANT}(u, x)$ 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 $x.\text{parent} = u$ 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])

```

8.8.1 Subindo violações

Na função $\text{ADDBLACK}(y, \textit{side})$, 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 \textit{side} têm um nó negro a menos que os caminhos passando por seu outro filho.

Seja x o filho de y na direção \textit{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 \textit{side} e do outro lado, respectivamente.

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

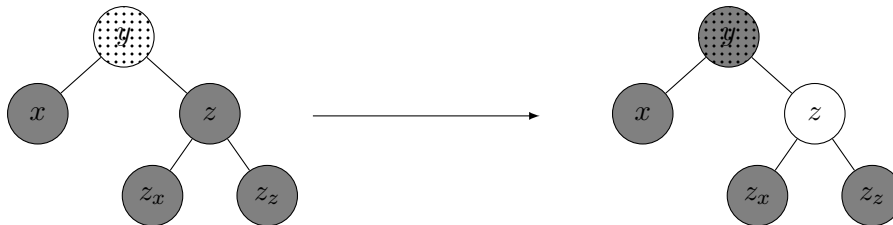


Figura 8.7: Aplicação do caso 1, não importa se x é filho direito ou esquerdo.

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 violada, e temos um nó negro a mais nos caminhos 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 na direção de y têm um nó preto a menos que os caminhos que vão na outra direção. Assim, recomeçamos o laço trocando y por seu pai e \textit{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.

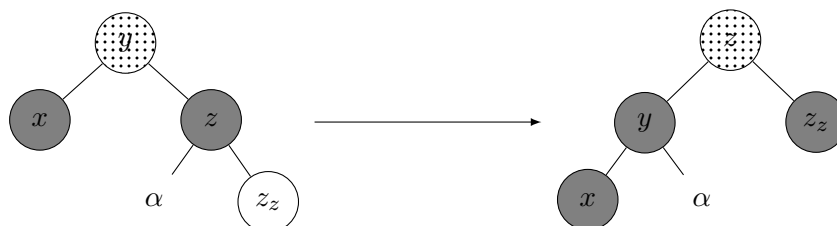


Figura 8.8: Aplicação do caso 2, assumindo que x é filho esquerdo.

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.

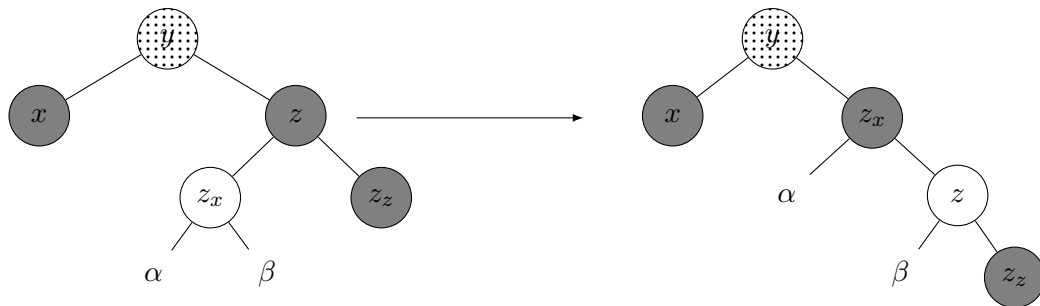


Figura 8.9: Aplicação do caso 3, assumindo que x é filho esquerdo.

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) 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.

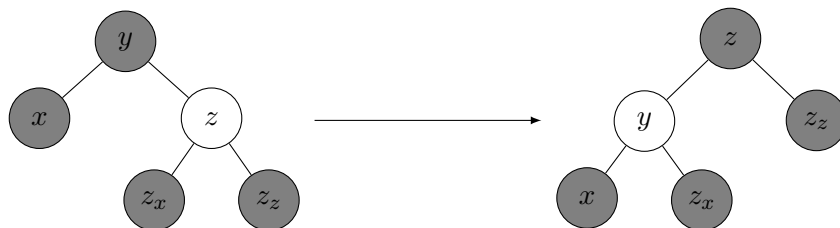


Figura 8.10: Aplicação do caso 4, assumindo que x é filho esquerdo.

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_x 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:    $y = \text{ACTIVE}(y)$  ▷ Versão mais atual de  $y$ .
3:   while  $y \neq \text{null}$  :
4:      $z = \text{CHILD}(y, \text{not } side)$ 
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:        $y = \text{ACTIVE}(y)$ 
9:        $z = \text{CHILD}(y, \text{not } side)$ 
10:     $z_x = \text{CHILD}(z, side)$ 
11:     $z_z = \text{CHILD}(z, \text{not } side)$ 
12:    if ( $z_x = \text{null}$  or not  $z_x.red$ ) and ( $z_z = \text{null}$  or not  $z_z.red$ ) : ▷ Caso 1.
13:       $z.red = \text{true}$ 
14:      if  $y = \text{roots}[\text{current}]$  or  $y.red$  :
15:         $y.red = \text{false}$ 
16:        break
17:      else
18:         $side = [\text{CHILD}(y.parent, 1) = y]$ 
19:         $y = y.parent$ 
20:    else
21:      if  $z_x \neq \text{null}$  and  $z_x.red$  : ▷ Arrumando caso 3 para caso 2.
22:        SWAP( $z.red$ ,  $z_x.red$ )
23:        ROTATE( $z$ ,  $side$ )
24:         $y = \text{ACTIVE}(y)$ 
25:         $z = \text{CHILD}(y, \text{not } side)$ 
26:         $z_z = \text{CHILD}(z, \text{not } side)$ 
27:        SWAP( $y.red$ ,  $z.red$ ) ▷ Caso 2.
28:         $z_z.red = \text{false}$ 
29:        ROTATE( $y$ , not  $side$ )
30:        break

```

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 segunda coluna da Tabela 8.1 mostra o consumo de tempo e espaço da implementação apresentada neste capítulo. Em ambas implementações, a operação `FIND` consome temporariamente espaço $\mathcal{O}(\lg n)$, devido a sua pilha de recursão.

Operação	Node copying	Funcional
<code>INSERT(value)</code>	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
<code>REMOVE(value)</code>	$\mathcal{O}(\lg n)/\mathcal{O}(1)$	$\mathcal{O}(\lg n)/\mathcal{O}(\lg n)$
<code>FIND(value)</code>	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$

Tabela 8.1: Comparação do consumo de tempo e espaço da implementação descrita 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.

Capítulo 9

Localização de ponto

Neste capítulo analisaremos uma variação do problema de localização de ponto (*point location*) [13], e mostraremos uma solução utilizando a ABB persistente descrita no Capítulo 8.

Dado um conjunto de polígonos $\{P_1, \dots, P_k\}$ 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 ao qual pertencem, ou pretos se não pertencem a nenhum dos polígonos.

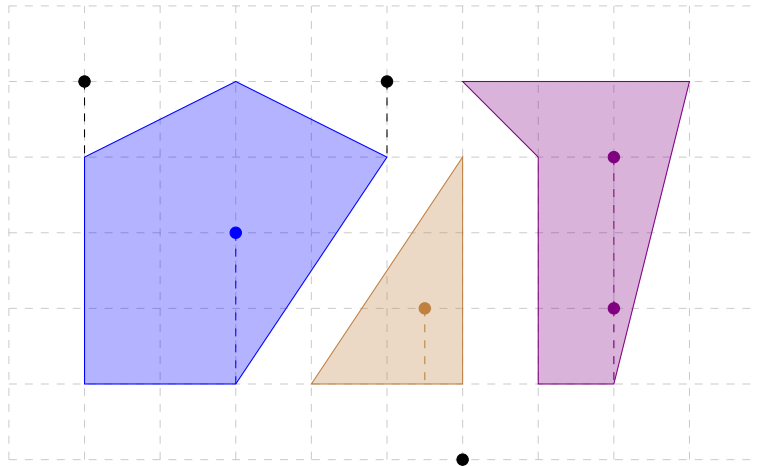


Figura 9.1: Exemplo do problema de localização de ponto.

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 $n := \sum_{i=1}^k |P_i|$, 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

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 da Figura 9.1, 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. Se sabemos em qual faixa está o ponto da consulta (o que pode ser determinado por uma busca binária pela coordenada x do ponto), é possível determinar o segmento diretamente abaixo deste ponto usando uma segunda busca binária nos segmentos presentes nesta faixa.

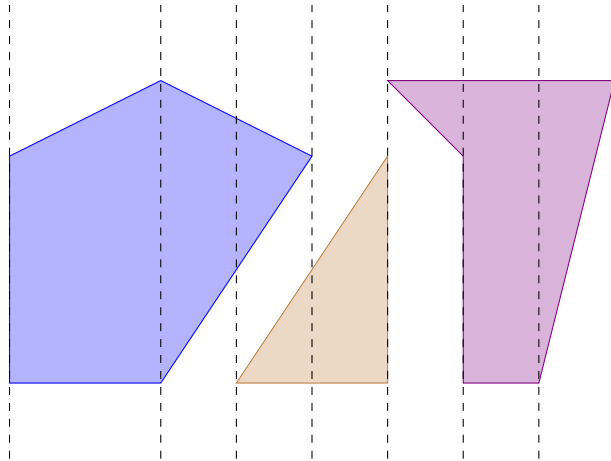


Figura 9.2: Partição do exemplo em faixas.

Se armazenarmos os segmentos de cada faixa de forma ingênua, isto se torna uma solução com complexidade de tempo $\langle \mathcal{O}(n^2 \lg n), \mathcal{O}(\lg n) \rangle$ e espaço $\mathcal{O}(n^2)$, 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 determinar todas as listas de segmentos de cada faixa usando apenas n adições e remoções de segmentos à faixa corrente.

Queremos manter os segmentos da faixa atual ordenados, e adicionar e remover segmentos ao longo do tempo, ao passar de uma faixa pra seguinte. Ademais, na fase das consultas, queremos realizar buscas na faixa em que o ponto de consulta se encontra para determinar o segmento diretamente abaixo desse ponto. Uma boa estrutura de dados para armazenar os segmentos de uma faixa é uma ABB, com os segmentos ordenados de baixo para cima. Numa ABB, operações de inserção, remoção e busca têm implementação eficiente. Note que há uma ordem total nos segmentos dentro de cada faixa, pois todos eles atravessam a faixa e não se intersectam no interior da faixa.

9.3 Conversão da solução 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 com os segmentos da faixa para determinar a resposta para cada uma das consultas.

Podemos, entretanto, usar uma ABB parcialmente persistente durante a varredura de forma que, na fase das consultas, dado um ponto p , podemos acessar a versão da ABB que corresponde à faixa que contém p e decidir se o ponto p está dentro de um dos polígonos. 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 $(\mathcal{O}(n \lg n), \mathcal{O}(\lg n))$, 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 $\{P_1, \dots, P_k\}$. 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 mais à esquerda do segmento. Usamos `SEGMENT(polygon, from, to, top)` para inicializar os campos de um segmento.

Na varredura, percorremos 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 da faixa 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 **ifs** 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( $\{P_1, \dots, P_k\}$ )
2:    $points = \{\}$  ▷ Vetor vazio
3:   for  $P_i \in \{P_1, \dots, P_k\}$  :
4:     for  $P_i^j \in P_i$  :
5:        $points.ADD(P_i^j)$ 
6:   Ordene  $points$  de forma que  $points[i] \leq points[j]$  se  $i < j$ . ▷  $\mathcal{O}(n \lg n)$ 
7:    $rbt =$  Árvore rubro-negra parcialmente persistente do Capítulo 8 inicialmente vazia.
8:    $slabs = \{\}$ 
9:    $slabs.ADD(((-\infty, 0), rbt.current))$ 
10:  for  $P_i^j \in points$  :
11:    if  $P_i^{j-1} < P_i^j$  : ▷ Segmento  $(j-1, j)$  vai para a direita
12:       $rbt.REMOVE(SEGMENT(i, P_i^{j-1}, P_i^j, \mathbf{true}))$  ▷  $\mathcal{O}(\lg n)$ 
13:    else
14:       $rbt.INSERT(SEGMENT(i, P_i^j, P_i^{j-1}, \mathbf{false}))$  ▷  $\mathcal{O}(\lg n)$ 
15:    if  $P_i^{j+1} > P_i^j$  : ▷ Segmento  $(j, j+1)$  vai para a direita
16:       $rbt.INSERT(SEGMENT(i, P_i^j, P_i^{j+1}, \mathbf{true}))$  ▷  $\mathcal{O}(\lg n)$ 
17:    else
18:       $rbt.REMOVE(SEGMENT(i, P_i^{j+1}, P_i^j, \mathbf{false}))$  ▷  $\mathcal{O}(\lg n)$ 
19:     $slabs.ADD((P_i^j, rbt.current))$ 

```

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, é de baixo para cima, e pode ser feita usando produto cruzado de vetores [12, Sec 1.3.2].

Por fim, a lista *slabs* é usada para armazenar a correspondência entre faixas e versões da ABB. Ela armazena, em ordem, o ponto que iniciou 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.

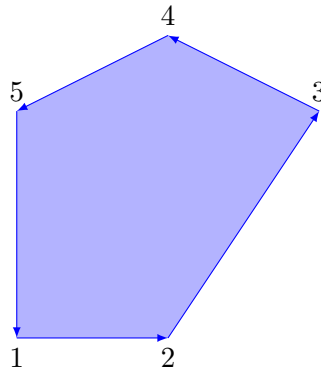


Figura 9.3: Exemplo do polígono dado em sentido anti-horário.

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  $q \leq p$ .           ▷ Busca binária  $\mathcal{O}(\lg n)$ 
3:   u = rbt.roots[current]
4:   below = null
5:   while u ≠ null :
6:     ▷ Se p está acima do segmento
7:     |   if CROSS(u.value.to - u.value.from, p - u.value.from) ≥ 0 :
8:     |   |   below = u.value
9:     |   |   u = rbt.CHILD(u, 1, current)
10:    |   else
11:    |   |   u = rbt.CHILD(u, 0, current)
12:    if below = null :
13:    |   return -1
14:    else if not below.top or CROSS(below.to - below.from, p - below.from) = 0 :
15:    |   return below.polygon
16:    else
17:    |   return -1

```

A linha 2 determina a faixa a qual p pertence, 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 Sedgwick e Wayne [14].

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.

As implementações de deque apresentadas mostram como reduzir a complexidade pode complicar bastante a teoria e implementação de uma estrutura. Como indicado na Tabela 6.1, para reduzir a complexidade de tempo de duas operações da deque, de logarítmico pra constante, foi necessário apresentar toda a teoria e código mais complexos 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 usa 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 parcialmente 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] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, NY, USA, 2nd edition, 1998.
- [13] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.
- [14] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.