Capítulo 13

Estruturas de Dados para Inteiros

Neste capítulo, voltamos ao problema de implementar um SSet. A diferença agora é que assumimos que os elementos armazenados no SSet são inteiros com w-bits. Ou seja, queremos implementar add(x), remove(x), e find(x) onde $x \in \{0, \ldots, 2^w - 1\}$. Não é muito difícil pensar em muitas aplicações em que os dados — ou pelo menos a chave que usamos para classificar os dados — é um número inteiro.

Discutiremos três estruturas de dados, cada uma com base nas idéias da anterior. A primeira estrutura, a BinaryTrie executa todas as três operações de um SSet em um tempo $O(w)$. Isso não é muito impressionante, pois qualquer subconjunto de $\{0, \ldots, 2^w - 1\}$ tem o tamanho $n \leq 2^w$, para que $\log n \leq w$. Todas as outras implementações de SSet discutidas neste livro realizam todas as operações em um tempo $O(\log n)$, para que sejam todas pelo menos tão rápidas quanto uma BinaryTrie.

A segunda estrutura, a XFastTrie, acelera a pesquisa em uma BinaryTrie usando hashing. Com esse aumento de velocidade, a operação find(x) é executada em um tempo $O(\log w)$. No entanto, as operações add(x) e remove(x) em uma XFastTrie ainda levam um tempo $O(w)$ e o espaço usado por uma XFastTrie é $O(n \cdot w)$.

A terceira estrutura de dados, YFastTrie, usa uma XFastTrie para armazenar apenas uma amostra de aproximadamente um de cada w elementos e armazena os elementos restantes em uma estrutura SSet padrão. Este truque reduz o tempo de execução de add(x) e remove(x) para $O(\log w)$ e diminui o espaço para $O(n)$.

As implementações usadas como exemplos neste capítulo podem ar-

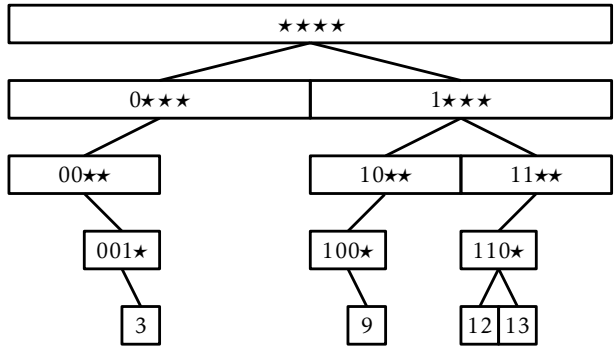

Figura 13.1: Os inteiros armazenados em uma trie binária são codificados como caminhos da raiz para a folha.

mazenar qualquer tipo de dados, desde que um inteiro possa ser associado a eles. Nos exemplos de código, a variável *ix* é sempre o valor inteiro associado a x, e o método IntValue(x) converte x em seu inteiro associado. No texto, entretanto, iremos simplesmente tratar x como se fosse um inteiro.

13.1 BinaryTrie: Uma árvore de busca digital

Uma BinaryTrie codifica um conjunto de inteiros de w bits em uma árvore binária. Todas as folhas da árvore têm profundidade w e cada número inteiro é codificado como um caminho da raiz para a folha. O caminho para o inteiro x vira à esquerda no nível i se o $i$ésimo bit mais significativo de x é um 0 e vira à direita se for 1. Figura 13.1 mostra um exemplo para o caso $w = 4$, no qual a trie armazena os inteiros 3(0011), 9(1001), 12(1100) e 13(1101).

Como o caminho de pesquisa para um valor x depende dos bits de x, será útil nomear os filhos de um nó, u, $u.child[0]$ (*left*) e $u.child[1]$ (*right*). Essas referências para os filhos na verdade servirão para dupla função. Como as folhas em uma trie binária não têm filhos, as referências são usadas para amarrar as folhas em uma lista duplamente encadeada. Para

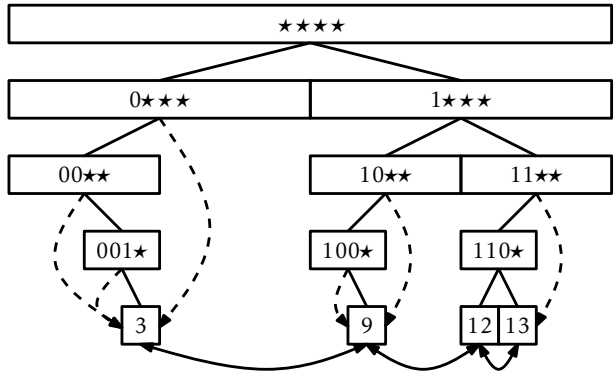

Figura 13.2: Uma BinaryTrie com ponteiros *jump* mostrados como arestas tracejadas curvas.

uma folha na trie binária $u.child[0]$ (*prev*) é o nó que vem antes de u na lista e $u.child[1]$ (*next*) é o nó que segue u na lista. Um nó especial, *dummy*, é usado antes do primeiro nó e depois do último nó da lista (ver Seção 3.2).

Cada nó, u, também contém um ponteiro adicional $u.jump$. Se o filho esquerdo de u estiver faltando, então $u.jump$ aponta para a menor folha na subárvore de u. Se o filho direito de u estiver faltando, $u.jump$ aponta para a maior folha na subárvore de u. Um exemplo de BinaryTrie, mostrando ponteiros *jump* e a lista duplamente encadeada nas folhas, é mostrado na Figura 13.2.

A operação find(x) em uma BinaryTrie é bastante simples. Tentamos seguir o caminho de pesquisa de x na trie. Se chegarmos a uma folha, encontramos x. Se chegarmos a um nó u onde não podemos prosseguir (porque u está faltando um filho), seguimos $u.jump$, que nos leva à menor folha maior que x ou à maior folha menor que x. Qual desses dois casos ocorre depende se em u está faltando seu filho esquerdo ou direito, respectivamente. No primeiro caso (em u está faltando seu filho esquerdo), encontramos o nó que desejamos. No último caso (em u está faltando seu filho direito), podemos usar a lista encadeada para chegar ao nó que desejamos. Cada um desses casos é ilustrado em Figura 13.3.

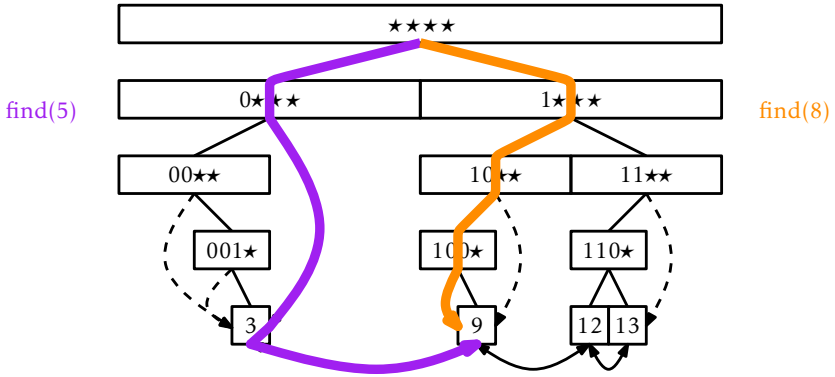

Figura 13.3: Os caminhos seguidos por find(5) e find(8).

```
find(x)
    ix ← int_value(x)
    u ← r
    i ← 0
    while i < w do
        c ← (ix ≫ w − i − 1) ∧ 1
        if u.child[c] = nil then break
        u ← u.child[c]
        i ← i + 1
    if i = w then return u.x # found it
    u ← [u.jump, u.jump.next][c]
    if u = dummy then return nil
    return u.x
```

O tempo de execução do método find(x) é dominado pelo tempo que leva para seguir um caminho da raiz para a folha, então ele é executado em tempo $O(w)$.

A operação add(x) em uma BinaryTrie também é bastante simples, mas tem muito trabalho a fazer:

1. Ele segue o caminho de busca por x até chegar a um nó u onde não pode mais prosseguir.

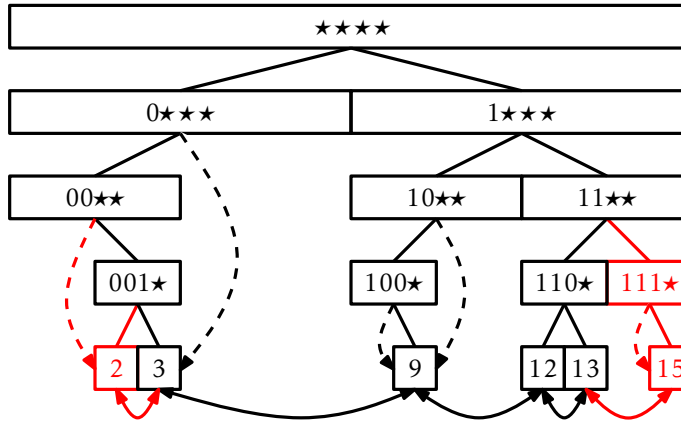

Figura 13.4: Adicionando os valores 2 e 15 à BinaryTrie na Figura 13.2.

2. Ele cria o restante do caminho de pesquisa de u para uma folha que contém x.

3. Ele adiciona o nó, u', contendo x à lista encadeada de folhas (tem acesso ao predecessor, *pred*, de u' na lista encadeada do ponteiro *jump* do último nó, u, encontrado durante a etapa 1.)

4. Ele caminha de volta no caminho de pesquisa para x, ajustando os ponteiros *jump* nos nós cujo ponteiro de *jump* deve agora apontar para x.

Uma adição é ilustrada na Figura 13.4.

```
add(x)
    ix ← int_value(x)
    u ← r
    # 1 - search for ix until falling out of the tree
    i ← 0
    while i < w do
        c ← (ix ≫ w − i − 1) ∧ 1
        if u.child[c] = nil then break
        u ← u.child[c]
```

```
    i ← i + 1
if i = w then return false # already contains x - abort
pred ← [u.jump.prev, u.jump][c]
u.jump ← nil # u will soon have two children
# 2 - add the path to ix
while i < w do
    c ← (ix ≫ w − i − 1) ∧ 1
    u.child[c] ← new_node()
    u.child[c].parent ← u
    u ← u.child[c]
    i ← i + 1
u.x ← x
# 3 - add u to the linked list
u.prev ← pred
u.next ← pred.next
u.prev.next ← u
u.next.prev ← u
# 4 - walk back up, updating jump pointers
v ← u.parent
while v ≠ nil do
    if (v.left = nil
            and (v.jump = nil or int_value(v.jump.x) > ix))
        or (v.right = nil
            and (v.jump = nil or int_value(v.jump.x) < ix))
        v.jump ← u
    v ← v.parent
n ← n + 1
return true
```

Este método executa uma caminhada pelo caminho de pesquisa de x e uma caminhada de volta para cima. Cada etapa dessas caminhadas leva um tempo constante, portanto o método add(x) é executado em um tempo $O(w)$.

A operação remove(x) desfaz o trabalho de add(x). Como add(x), ele tem muito trabalho a fazer:

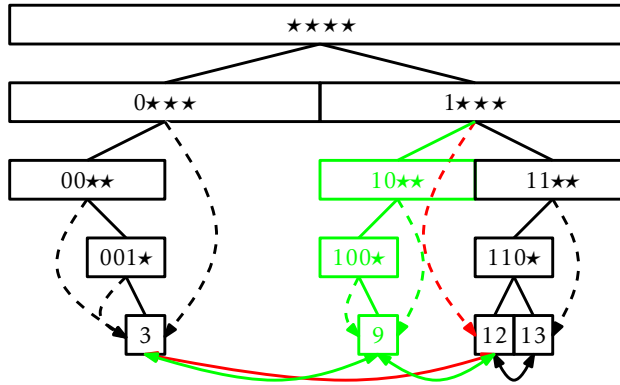

Figura 13.5: Removendo o valor 9 da BinaryTrie na Figura 13.2.

1. Segue o caminho de busca por x até chegar à folha, u, que contém x.
2. Ele remove u da lista duplamente encadeada.
3. Ele exclui u e, em seguida, retorna ao caminho de pesquisa de x excluindo nós até chegar a um nó v que tem um filho que não está no caminho de pesquisa de x.
4. Ele sobe de v para a raiz, atualizando quaisquer ponteiros *jump* que apontam para u.

Uma remoção é ilustrada na Figura 13.5.

```
remove(x)
    ix ← int_value(x)
    u ← r
    # 1 - find leaf, u, that contains x
    i ← 0
    while i < w do
        c ← (ix ≫ w − i − 1) ∧ 1
        if u.child[c] = nil then return false
        u ← u.child[c]
        i ← i + 1
```

```
    # 2 - remove u from linked list
    u.prev.next ← u.next
    u.next.prev ← u.prev
    v ← u
    # 3 - delete nodes on path to u
    for i in w − 1, w − 2, w − 3, ..., 0 do
        c ← (ix ≫ w − i − 1) ∧ 1
        v ← v.parent
        v.child[c] ← nil
        if v.child[1 − c] ≠ nil then break
    # 4 - update jump pointers
    pred ← u.prev
    succ ← u.next
    v.jump ← [pred, succ][v.left = nil]
    v ← v.parent
    while v ≠ nil do
        if v.jump = u then
            v.jump ← [pred, succ][v.left = nil]
        v ← v.parent
    n ← n − 1
    return true
```

Teorema 13.1. *Uma BinaryTrie implementa a interface SSet para inteiros de w-bits. Uma BinaryTrie suporta as operações* add(x), remove(x) *e* find(x) *em um tempo $O(w)$ por operação. O espaço usado por uma BinaryTrie que armazena n valores é $O(n \cdot w)$.*

13.2 XFastTrie: Pesquisando em tempo duplamente logarítmico

O desempenho da estrutura BinaryTrie não é muito impressionante. O número de elementos, n, armazenados na estrutura é no máximo 2^w, então $\log n \leq w$. Em outras palavras, qualquer uma das estruturas SSet baseadas em comparação descritas em outras partes deste livro são pelo menos tão eficientes quanto BinaryTrie e não estão restritas a armazenar

apenas inteiros.

Em seguida, descrevemos a XFastTrie, que é apenas uma BinaryTrie com $w+1$ tabelas hash — uma para cada nível do teste. Essas tabelas hash são usadas para acelerar a operação find(x) para um tempo $O(\log w)$. Lembre-se de que a operação find(x) em uma BinaryTrie está quase completa quando alcançamos um nó, u, onde o caminho de pesquisa para x gostaria de prosseguir para $u.right$ (ou $u.left$) mas u não tem filho direito (respectivamente, esquerdo). Neste ponto, a pesquisa usa $u.jump$ para pular para uma folha, v, da BinaryTrie e retornar v ou seu sucessor na lista encadeada de folhas. Uma XFastTrie acelera o processo de pesquisa usando a pesquisa binária nos níveis da trie para localizar o nó u.

Para usar a pesquisa binária, precisamos determinar se o nó u que estamos procurando está acima de um determinado nível, i, ou se u está no nível ou abaixo do i. Esta informação é fornecida pelos bits i de ordem mais alta na representação binária de x; esses bits determinam o caminho de pesquisa que x leva da raiz ao nível i. Para um exemplo, consulte Figura 13.6; nesta figura, o último nó, u, no caminho de pesquisa para 14 (cuja representação binária é 1110) é o nó rotulado 11$\star\star$ no nível 2 porque não há nenhum nó rotulado 111$\star$ no nível 3. Portanto, podemos rotular cada nó no nível i com um número inteiro de i bits. Então, o nó u que estamos procurando estaria no nível i ou abaixo se e somente se houvesse um nó no nível i cujo rótulo corresponda aos bits i de ordem mais alta de x.

Em uma XFastTrie, armazenamos, para cada $i \in \{0,\ldots,w\}$, todos os nós no nível i em um USet, $t[i]$, que é implementado como uma tabela hash (Capítulo 5). Usar este USet nos permite verificar em tempo esperado constante se há um nó no nível i cujo rótulo corresponde aos bits i de ordem mais alta de x. Na verdade, podemos até encontrar este nó usando $t[i]$.find$(x \gg (w-i))$

As tabelas de hash $t[0],\ldots,t[w]$ nos permitem usar a pesquisa binária para encontrar u. Inicialmente, sabemos que u está em algum nível i com $0 \le i < w+1$. Portanto, inicializamos $\ell = 0$ e $h = w+1$ e repetidamente olhamos para a tabela hash $t[i]$, onde $i = \lfloor(\ell+h)/2\rfloor$. Se $t[i]$ contém um nó cujo rótulo corresponde aos bits i de ordem superior de x, então definimos $\ell \leftarrow i$ (u está no nível ou abaixo de i); caso contrário, definimos $h \leftarrow i$ (u está acima do nível i). Este processo termina quando $h-\ell \le 1$, caso em que

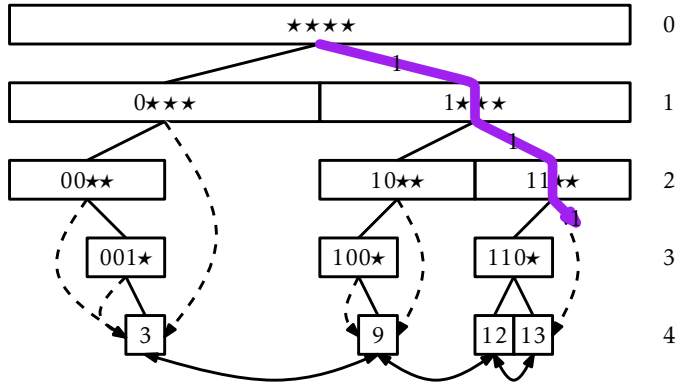

Figura 13.6: Como não há nenhum nó rotulado como 111★, o caminho de pesquisa para 14 (1110) termina no nó rotulado 11★★.

determinamos que u está no nível ℓ. Em seguida, completamos a operação find(x) usando $u.jump$ e a lista duplamente encadeada de folhas.

```
find(x)
    ix ← int_value(x)
    ℓ, h ← 0, w + 1
    u ← r
    q ← new_node()
    while h − ℓ > 1 do
        i ← (ℓ + h)/2
        q.prefix ← ix ≫ w − i
        v ← t[i].find(q)
        if v = nil then
            h ← i
        else
            u ← v
            ℓ ← i
    if ℓ = w then return u.x
    c ← ix ≫ (w − ℓ − 1) ∧ 1
    pred ← [u.jump.prev, u.jump][c]
    if pred.next = nil then return nil
```

return *pred.next.x*

Cada iteração do loop **while** no método acima diminui $h - \ell$ por aproximadamente um fator de dois, então este loop encontra u após $O(\log w)$ iterações. Cada iteração executa uma quantidade constante de trabalho e uma operação find(x) em um USet, que leva um tempo esperado constante. O trabalho restante leva apenas um tempo constante, portanto o método find(x) em uma XFastTrie leva apenas um tempo esperado de $O(\log w)$.

Os métodos add(x) e remove(x) para uma XFastTrie são quase idênticos aos mesmos métodos em uma BinaryTrie. As únicas modificações são para gerenciar as tabelas hash $t[0],\ldots,t[w]$. Durante a operação add(x), quando um novo nó é criado no nível i, este nó é adicionado a $t[i]$. Durante uma operação remove(x), quando um nó é removido do nível i, este nó é removido de $t[i]$. Como adicionar e remover de uma tabela hash leva um tempo esperado constante, isso não aumenta os tempos de execução de add(x) e remove(x) por mais de um fator constante. Omitimos uma listagem de código para add(x) e remove(x), pois o código é quase idêntico à (longa) listagem de código já fornecida para os mesmos métodos em uma BinaryTrie.

O teorema a seguir resume o desempenho de uma XFastTrie:

Teorema 13.2. *Uma XFastTrie implementa a interface SSet para inteiros com w-bits. Uma XFastTrie suporta as operações*

- add(x) *e* remove(x) *em tempo esperado de $O(w)$ por operação e*
- find(x) *em tempo esperado de $O(\log w)$ por operação.*

O espaço usado por uma XFastTrie que armazena n valores é $O(n \cdot w)$.

13.3 YFastTrie: Um SSet de tempo duplamente logarítmico

A XFastTrie é uma grande – até exponencial – melhoria em relação à BinaryTrie em termos de tempo de consulta, mas as operações add(x) e remove(x) ainda não são terrivelmente rápidas. Além disso, o uso de espaço, $O(n \cdot w)$, é maior do que as outras implementações de SSet descritas

neste livro, que usam $O(n)$ espaço. Esses dois problemas estão relacionados; se n add(x) operações constroem uma estrutura de tamanho $n \cdot w$, então a operação add(x) requer pelo menos na ordem de w tempo (e espaço) por operação.

A YFastTrie, discutida a seguir, melhora simultaneamente o espaço e a velocidade das XFastTries. Uma YFastTrie usa uma XFastTrie, *xft*, mas armazena apenas $O(n/w)$ valores em *xft*. Desta forma, o espaço total usado por *xft* é apenas $O(n)$. Além disso, apenas uma de cada w add(x) ou remove(x) operações em YFastTrie resulta em uma operação add(x) ou remove(x) em *xft* . Fazendo isso, o custo médio incorrido por chamadas para *xft* operações add(x) e remove(x) é apenas constante.

A pergunta óbvia é: Se *xft* armazena apenas n/w elementos, para onde vão os $n(1-1/w)$ elementos restantes? Esses elementos se movem para *estruturas secundárias*, neste caso, uma versão estendida de treaps(Seção 7.2). Existem aproximadamente n/w dessas estruturas secundárias, portanto, em média, cada uma delas armazena $O(w)$ itens. Treaps suportam operações de SSet em tempo logarítmico, então as operações nesses treaps serão executadas em tempo $O(\log w)$, conforme necessário.

Mais concretamente, uma YFastTrie contém uma XFastTrie, *xft*, que contém uma amostra aleatória dos dados, onde cada elemento aparece na amostra independentemente com probabilidade $1/w$. Por conveniência, o valor $2^w - 1$, está sempre contido em *xft*. Faça $x_0 < x_1 < \cdots < x_{k-1}$ denotar os elementos armazenados em *xft*. Associado a cada elemento, x_i, está um treap, t_i, que armazena todos os valores no intervalo $x_{i-1}+1,\ldots,x_i$. Isso é ilustrado em Figura 13.7.

A operação find(x) em uma YFastTrie é bastante fácil. Procuramos por x em *xft* e encontramos algum valor x_i associado ao treap t_i. Em seguida, usamos o método da treap find(x) em t_i para responder à consulta. Todo o método é de uma linha:

```
find(x)
    return xft.find(Pair(int_value(x)))[1].find(x)
```

A primeira operação find(x) (em *xft*) leva um tempo $O(\log w)$. A segunda operação find(x) (em uma treap) leva um tempo $O(\log r)$, onde r

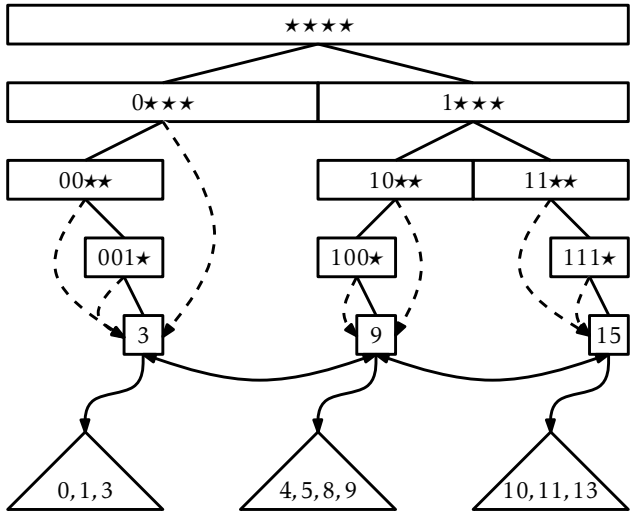

Figura 13.7: Uma YFastTrie contendo os valores 0, 1, 3, 4, 6, 8, 9, 10, 11, e 13.

é o tamanho da treap. Posteriormente nesta seção, mostraremos que o tamanho esperado da treap é $O(w)$, de modo que esta operação leva um tempo $O(\log w)$.[1]

Adicionar um elemento a uma YFastTrie também é bastante simples — na maioria das vezes. O método add(x) chama xft.find(x) para localizar a treap, t, na qual x deve ser inserido. Em seguida, chama t.add(x) para adicionar x a t. Nesse ponto, ele lança uma moeda tendenciosa que sai cara com probabilidade $1/w$ e coroa com probabilidade $1 - 1/w$. Se esta moeda der cara, então x será adicionado a xft.

É aqui que as coisas ficam um pouco mais complicadas. Quando x é adicionado à xft, a treap t precisa ser dividida em duas treaps, t_1 e t'. A treap t_1 contém todos os valores menores ou iguais a x; t' é a treap original, t, com os elementos de t_1 removidos. Feito isso, adicionamos o par (x, t_1) a xft. A Figura 13.8 mostra um exemplo.

```
add(x)
    ix ← int_value(x)
```

[1] Esta é uma aplicação da *Desigualdade de Jensen*: Se $\mathrm{E}[r] = w$, então $\mathrm{E}[\log r] \le \log w$.

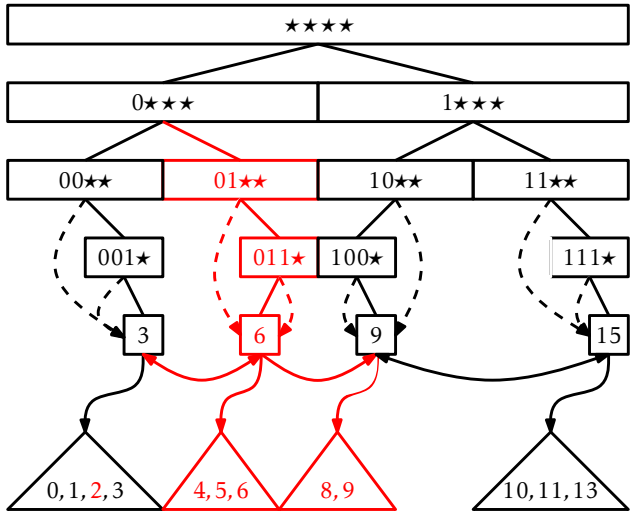

Figura 13.8: Adicionando os valores 2 e 6 a uma YFastTrie. O sorteio de 6 deu cara, então 6 foi adicionado à *xft* e a treap contendo $4, 5, 6, 8, 9$ foi dividida.

```
t ← xft.find(Pair(ix))[1]
if t.add(x) then
    n ← n + 1
    if random.randrange(w) = 0 then
        t1 ← t.split(x)
        xft.add(Pair(ix, t1))
    return true
return false
```

Adicionar x a t leva um tempo $O(\log w)$. Exercício 7.12 mostra que dividir t em t_1 e t' também pode ser feito em um tempo esperado de $O(\log w)$. Adicionar o par (x, t_1) a *xft* leva um tempo $O(w)$, mas só acontece com a probabilidade $1/w$. Portanto, o tempo de execução esperado da operação add(x) é

$$O(\log w) + \frac{1}{w}O(w) = O(\log w) \ .$$

O método remove(x) desfaz o trabalho executado por add(x). Usamos

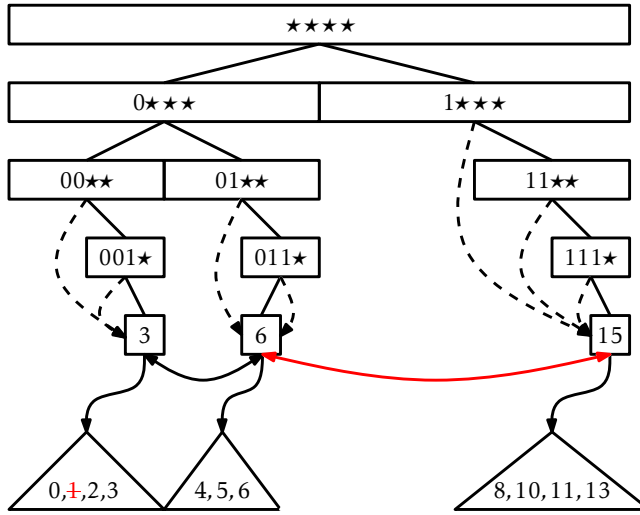

Figura 13.9: Removendo os valores 1 e 9 de uma YFastTrie na Figura 13.8.

xft para encontrar a folha, u, em *xft* que contém a resposta para *xft*.find(x). De u, obtemos a treap, t, contendo x e removemos x de t. Se x também foi armazenado em *xft* (e x não é igual a $2^w - 1$), então removemos x de *xft* e adicionamos os elementos da treap de x à treap, t_2, que é armazenada pelo sucessor de u na lista encadeada. Isso é ilustrado em Figura 13.9.

```
remove(x)
    ix ← int_value(x)
    u ← xft.find_node(ix)
    ret ← u.x[1].remove(x)
    if ret then n ← n − 1
    if u.x[0] = ix and ix ≠ 2^w − 1 then
        t2 ← u.next.x[1]
        t2.absorb(u.x[1])
        xft.remove(u.x)
    return ret
```

Encontrar o nó u em *xft* leva um tempo esperado de $O(\log w)$. Remover

x de t leva um tempo esperado $O(\log w)$. Novamente, o Exercício 7.12 mostra que mesclar todos os elementos de t em t_2 pode ser feito em tempo $O(\log w)$. Se necessário, remover x de xft leva um tempo $O(w)$, mas x só está contido em xft com probabilidade $1/w$. Portanto, o tempo esperado para remover um elemento de uma YFastTrie é $O(\log w)$.

No início da discussão, atrasamos a discussão sobre os tamanhos das treaps nesta estrutura para mais tarde. Antes de terminar este capítulo, provamos o resultado de que precisamos.

Lema 13.1. *Seja x um inteiro armazenado em uma YFastTrie e seja n_x o número de elementos na treap, t, que contém x. Então* $\mathrm{E}[n_x] \le 2w - 1$.

Demonstração. Referir-se à Figura 13.10. Seja $x_1 < x_2 < \cdots < x_i = x < x_{i+1} < \cdots < x_n$ os elementos armazenados na YFastTrie. A treap t contém alguns elementos maiores ou iguais a x. Esses são $x_i, x_{i+1}, \ldots, x_{i+j-1}$, onde x_{i+j-1} é o único desses elementos em que o lance de moeda tendencioso realizado no método add(x) deu cara. Em outras palavras, $\mathrm{E}[j]$ é igual ao número esperado de lançamentos tendenciosos de moeda necessários para obter as primeiras caras.[2] Cada sorteio é independente e sai cara com probabilidade $1/w$, então $\mathrm{E}[j] \le w$. (Veja Lema 4.2 para uma análise disto para o caso $w = 2$.)

Da mesma forma, os elementos de t menores que x são $x_{i-1}, \ldots, x_{i-k}$ onde todos esses k lançamentos de moeda resultam em coroa e o lançamento de x_{i-k-1} dá cara. Portanto, $\mathrm{E}[k] \le w-1$, já que este é o mesmo experimento de lançamento de moeda considerado no parágrafo anterior, mas aquele em que o último lançamento não é contado. Em resumo, $n_x = j + k$, então

$$\mathrm{E}[n_x] = \mathrm{E}[j + k] = \mathrm{E}[j] + \mathrm{E}[k] \le 2w - 1 \ . \qquad \square$$

Lema 13.1 foi a última peça na prova do seguinte teorema, que resume o desempenho da YFastTrie:

Teorema 13.3. *Uma YFastTrie implementa a interface SSet para inteiros de w-bits. Uma YFastTrie suporta as operações* add(x), remove(x) *e* find(x) *em tempo esperado de $O(\log w)$ por operação. O espaço usado por uma YFastTrie que armazena n valores é $O(n + w)$.*

[2] Esta análise ignora o fato de que j nunca excede $n-i+1$. No entanto, isso apenas diminui $\mathrm{E}[j]$, então o limite superior ainda se mantém.

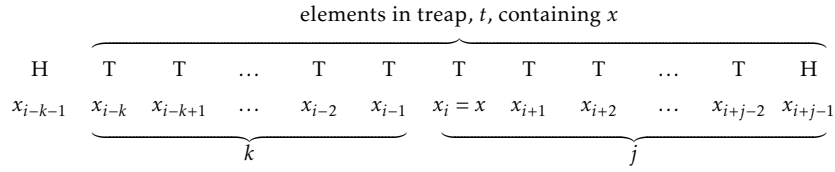

Figura 13.10: O número de elementos na treap t contendo x é determinado por dois experimentos de lançamento de moeda.

O termo w no requisito de espaço vem do fato de que *xft* sempre armazena o valor $2^w - 1$. A implementação pode ser modificada (às custas de adicionar alguns casos extras ao código) para que seja desnecessário armazenar esse valor. Neste caso, o requisito de espaço no teorema torna-se $O(n)$.

13.4 Discussão e Exercícios

A primeira estrutura de dados a fornecer operações add(x), remove(x) e find(x) em um tempo $O(\log w)$ foi proposta por van Emde Boas e desde então tornou-se conhecida como o *árvore van Emde Boas* (ou *estratificada*) [72]. A estrutura original de van Emde Boas tinha tamanho 2^w, tornando-a impraticável para números inteiros grandes.

As estruturas de dados XFastTrie e YFastTrie foram descobertas por Willard [75]. A estrutura XFastTrie está intimamente relacionada às árvores van Emde Boas; por exemplo, as tabelas de hash em uma XFastTrie substituem arrays em uma árvore van Emde Boas. Ou seja, em vez de armazenar a tabela de hash $t[i]$, uma árvore van Emde Boas armazena uma matriz de comprimento 2^i.

Outra estrutura para armazenar inteiros são as árvores de fusão de Fredman e Willard [32]. Esta estrutura pode armazenar n inteiros de w bits em um espaço $O(n)$ de modo que a operação find(x) execute em um tempo $O((\log n)/(\log w))$. Usando uma árvore de fusão quando $\log w > \sqrt{\log n}$ e uma YFastTrie quando $\log w \leq \sqrt{\log n}$, obtém-se uma estrutura de dados de espaço $O(n)$ que pode implementar a operação find(x) em tempo $O(\sqrt{\log n})$. Resultados recentes de limite inferior de Pătraşcu e

Thorup [57] mostram que esses resultados são mais ou menos ideais, pelo menos para estruturas que usam apenas espaço $O(n)$.

Exercício 13.1. Projete e implemente uma versão simplificada de uma BinaryTrie que não tenha uma lista encadeada ou ponteiros de salto, mas para o qual find(x) ainda é executado em tempo $O(w)$.

Exercício 13.2. Projete e implemente uma implementação simplificada de uma XFastTrie que não use um teste binário. Em vez disso, sua implementação deve armazenar tudo em uma lista duplamente encadeada e $w+1$ tabelas de hash.

Exercício 13.3. Podemos pensar na BinaryTrie como uma estrutura que armazena sequências de bits de comprimento w de forma que cada sequência de bits seja representada como um caminho da raiz para folha. Estenda essa ideia em uma implementação SSet que armazena strings de comprimento variável e implementa add(s), remove(s) e find(s) no tempo proporcional ao comprimento de s.
Dica: Cada nó em sua estrutura de dados deve armazenar uma tabela hash que é indexada por valores de caractere.

Exercício 13.4. Para um inteiro $x \in \{0,\ldots 2^w-1\}$, faça $d(x)$ denotar a diferença entre x e o valor retornado por find(x) [se find(x) retorna *nil*, então defina $d(x)$ como 2^w]. Por exemplo, se find(23) retorna 43, então $d(23) = 20$.

1. Projete e implemente uma versão modificada da operação find(x) em uma XFastTrie executada em tempo esperado de $O(1+\log d(x))$. Dica: a tabela hash $t[w]$ contém todos os valores, x, de forma que $d(x) = 0$, então esse seria um bom lugar para começar.

2. Projete e implemente uma versão modificada da operação find(x) em uma XFastTrie executada em um tempo esperado de $O(1+\log\log d(x))$.