Operações Eficientes em Arrays Dinâmicos

Bruno Monteiro
Orientador: Vinicius dos Santos
Co-orientador: Mauricio Collares

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

setembro de 2021

Operações Eficientes em Arrays Dinâmicos

Bruno Monteiro
Orientador: Vinicius dos Santos
Co-orientador: Mauricio Collares

Resumo. *Representação de conjuntos é um problema extensivamente estudado. Porém, soluções canônicas com árvores binárias de busca não são trivialmente capazes de unir dois conjuntos eficientemente, ao mesmo tempo permitindo a separação de conjuntos por um dado valor. Estudamos uma estrutura de dados capaz de representar conjuntos de inteiros não-negativos e efetuar tais operações em tempo $\mathcal{O}(\log U)$ amortizado, se U é o maior valor que podemos representar. Propomos também um novo tipo abstrato de dados que chamamos de* BLOCK-SORTED ARRAY*, e mostramos que com ele é pode-se arrays de inteiros não-negativos com operações de separar um array em dois em uma dada posição, concatenar, reverter e ordenar arrays, em tempo $\mathcal{O}(\log n + \log U)$ amortizado por operação, sendo n o número total de elementos representados e U o valor máximo representado. Por fim, definimos uma nova operação, chamada de* PARTITION*, que se trata de remover os k menores valores de um array, de forma ordenada. Mostramos uma implementação dessa operação usando o* BLOCK-SORTED ARRAY*, e conjecturamos que ela é efetuada em tempo sub-linear amortizado.*

Abstract. *Set representation is an extensively studied problem. However, canonical solutions using binary search trees are not trivially capable of merging two sets efficiently, while allowing splitting of the sets by a given value. We study a data structure capable of representing non-negative integer sets and applying these operations in $\mathcal{O}(\log U)$ amortized time, with U being the largest element we can represent. We propose a new abstract data type that we call* BLOCK-SORTED ARRAY*, and we show that it is capable of representing non-negative integers arrays while handling the operations of splitting an array into two at a given position, concatenating, reversing and sorting arrays in $\mathcal{O}(\log n + \log U)$ amortized time per operation, with n being the total number of represented values and U being the largest represented value. Finally, we define a new operation called* PARTITION*, that means removing the k smallest values from an array, in sorted order. We show an implementation of this operation using the* BLOCK-SORTED ARRAY*, and we conjecture that it takes only sub-linear amortized time.*

Sumário

Capítulo 1

Representação de Conjuntos

1.1. Introdução

Um problema clássico na Computação é o de representar conjuntos eficientemente. Mais especificamente, gostaríamos de ser capazes de inserir, remover e buscar por itens em tempo sub-linear ($o(n)$, para n itens). Assumimos que existe uma Ordem Total entre os itens.

A solução clássica para esse problema envolve Árvores Binárias de Busca (ABB), em que os itens são armazenados em uma árvore enraizada, de forma que cada vértice contenha um item e no máximo dois filhos (um filho *esquerda* e um *direita*) [1]. A propriedade que ABBs satisfazem é que o item armazenado em um vértice é maior que todos os itens da sub-árvore de seu filho *esquerda* e menor que todas os itens da sub-árvore de seu filho *direita*.

Utilizando essa representação combinada com alguma estratégia para manter a árvore "balanceada", é possível representar um conjunto de itens com uma Ordem Total tal que as operações descritas são realizadas em tempo $\mathcal{O}(\log n)$ [2, 3].

Nós estudaremos um caso particular desse problema que é bastante comum: quando o conjunto de itens possíveis são os naturais até um certo valor U, ou seja, queremos representar subconjuntos de $\{0, 1, 2, 3, \ldots, U\}$. Nesse cenário, veremos que tipo de operações podem ser realizadas eficientemente, em tempo logarítmico em função de U.

1.2. Trabalhos Relacionados

Um problema de interesse é o *Mergeable Dictionary*, que se trata de manter uma coleção dinâmica de subconjuntos de $\{0, 1, 2, 3, \ldots, U\}$, sujeita às seguintes operações:

1. MAKESET(): retorna um conjunto vazio.
2. INSERT(S, x): insere o valor x em S, com $0 \leq x \leq U$.
3. DELETE(S, x): remove o valor x de S, se $x \in S$.
4. SEARCH(S, x): retorna o menor elemento em S que não é menor que x. Se não existe tal elemento, retorna -1.
5. SPLIT(S, x): retorna um par de conjuntos (A, B), tal que $A = \{y \in S : y < x\}$ e $B = S \setminus A$. O conjunto S é destruído.
6. MERGE(A, B): retorna um conjunto $S = A \cup B$. Os conjuntos A e B são destruídos.

A maioria das ABBs suportam as operações 1, 2, 3, 4 e 5 em tempo logarítmico em função do número de elementos representados, e suportam a operação 6 apenas se o maior elemento de A é menor que o menor elemento de B [3] (operação 6 **restrita**).

Em 1998, Martin Farach e Mikkel Thorup mostraram um algoritmo chamado *segment merge* que, a partir de uma ABB que suporta a operação 6 restrita, faz a operação 6 em tempo $\mathcal{O}(\log n \log U)$ amortizado, ou seja, qualquer sequência de q operações 1, 2, 3, 4, 5 e 6 pode ser feita em tempo $\mathcal{O}(q \log n \log U)$ no pior caso [4]. Esse algoritmo se baseia em quebrar os conjuntos A e B que se deseja juntar em blocos tal que a operação 6 restrita possa ser aplicada repetidamente, como ilustrado na Figura 1.1.

$$A = \{1, 3, 7, 9, 13, 14\}$$
$$B = \{5, 6, 8, 10, 12, 20\}$$

	A_1	B_1	A_2	B_2	A_3	B_3	A_4	B_4
A	1 3		7		9		13 14	
B		5 6		8		10 12		20

Figura 1.1. Exemplo do algoritmo *segment merge* proposto em [4].

Em 2010, John Iacono e Özgür Özkan criaram uma estrutura capaz de realizar a operação 6 em tempo $\mathcal{O}(\log U)$ amortizado [5], porém ela é bastante complicada. Uma solução mais simples foi proposta em 2016 por Adam Karczmarz [6], que é a estrutura principal que estudaremos. Ela é capaz de realizar as operações 1, 2, 3, 4, 5 em tempo $\mathcal{O}(\log U)$ amortizado e no pior caso, e a operação 6 em $\mathcal{O}(\log U)$ amortizado.

Além disso, qualquer estrutura de dados requer tempo $\Omega(\log n)$ para realizar pelo menos uma dentre as operações 4, 5 e 6 [6].

1.3. A Estrutura de Dados

A ideia da estrutura de dados é representar os elementos como uma *trie* [1] sobre a representação binária dos elementos do conjunto. Assim, precisamos de $\lceil \log U \rceil$ *bits* para representar cada um dos elementos, então a *trie* possui altura $\mathcal{O}(\log U)$ (ver Figura 1.2).

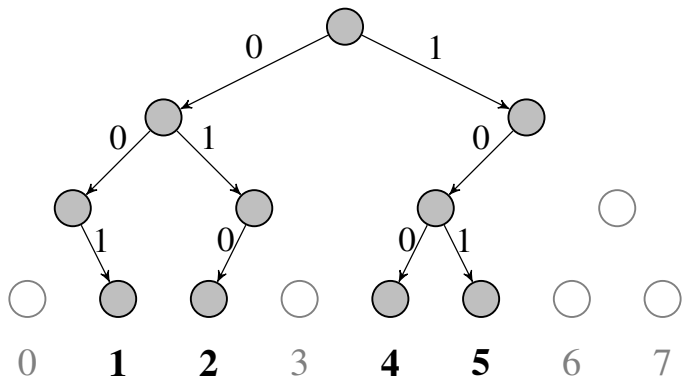


Figura 1.2. Exemplo de uma *trie* representando $S = \{1, 2, 4, 5\}$, para $U = 7$.

As operações INSERT e ERASE são trivialmente feitas em $\mathcal{O}(\log U)$, pois são apenas inserções e remoções em uma *trie*. SEARCH(S, x) também é simples, primeiro

descemos até o maior prefixo dos *bits* de x que existe na *trie* relativa a S. Se chegarmos a uma folha, retornamos x. Caso contrário subimos na *trie* até encontrar o primeiro nó tal que ele possua um filho da direita e acabamos de subir pela esquerda, e em seguida descemos até a folha mais a esquerda da sub-árvore desse nó. Isso é feito em $\mathcal{O}(\log U)$.

Para fazer a operação SPLIT(S, x), precisamos criar duas *tries*, A e B, tal que $A = \{y \in S : y < x\}$ e $B = S \setminus A$. Primeiro descemos até a folha relativa ao menor elemento em S que não é menor que x em $\mathcal{O}(\log U)$, como é feito na operação SEARCH. Agora queremos colocar em A todos os nós de caminhos até folhas menores que a folha atual. Para isso, vamos subindo na *trie*, até o primeiro momento em que subimos de um filho da direita. A partir desse momento, passamos a copiar os nós que visitamos para A, e também mover todas as sub-árvores da esquerda para A, sempre que subimos da direita (ver Figura 1.3). O nós restantes na *trie* inicial ao final do algoritmo são atribuídos a B. Esse processo custa $\mathcal{O}(\log U)$.

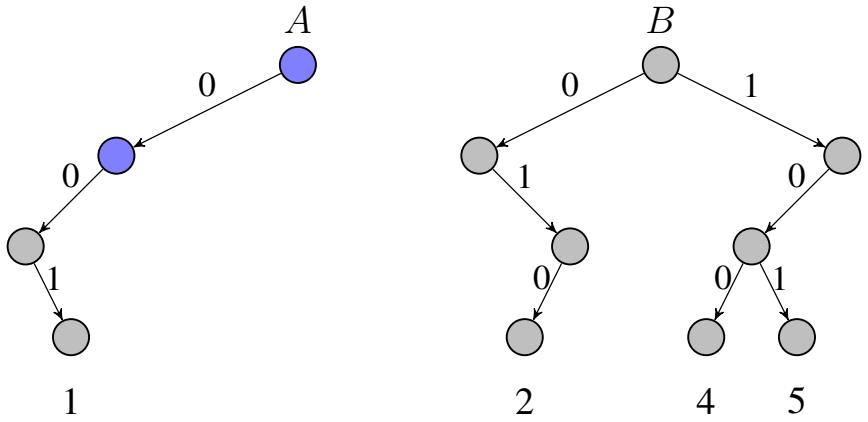


Figura 1.3. Resultado de aplicar a SPLIT(S, 2) na instância representada na Figura Figura 1.2. Os nós em azul foram criados (copiados) durante a operação.

Podemos fazer a operação MERGE(A, B) recursivamente: seja `merge_trie(a, b)` a função recursiva que recebe dois nós e retorna a raiz da *trie* que representa a união das *tries* cujas raízes são `a` e `b`. Podemos implementar a operação como descrito no Algoritmo 1. Assumimos que os campos `left` e `right` representam, respectivamente, os filhos esquerda e direita do nó, e que o campo `root` representa a raiz da *trie*.

Temos que a operação MERGE(A, B) tem pior caso $\Theta(\min(|A|, |B|) \log U)$, que é $\Theta(n \log U)$ para n elementos representados. Porém, o custo **amortizado** da operação é $\mathcal{O}(1)$, como provaremos no Teorema 1.

1.4. Análise de Complexidade Amortizada

Nesta seção, vamos explicar brevemente como funciona análise de complexidade amortizada por função potencial. Ao fazer análise de complexidade de operações em estruturas de dados, certas vezes uma operação pode ter pior caso ruim, porém isso não pode acontecer sempre: ao fazer uma operação custosa, as operações seguintes são mais baratas. Para isso serve a análise de complexidade *amortizada*: queremos medir o custo "médio" de uma operação no pior caso.

Algoritmo 1 MERGE(A, B)

```
função MERGE_TRIE(a, b)
    se a = NULL então
        retorne b
    se b = NULL então
        retorne a
    a.left = MERGE_TRIE(a.left, b.left)
    a.right = MERGE_TRIE(a.right, b.right)
    apague b
    retorne a

S ← MAKESET()
S.root = MERGE_TRIE(A.root, B.root)
A.root = B.root = NULL
retorne S
```

É importante ressaltar que a análise de complexidade amortizada ainda se trata do pior caso, ou seja, não fazemos nenhuma suposição sobre a entrada ou a ordem das operações. O objetivo é definir o custo amortizado de uma operação de tal forma que qualquer sequência de operações tenha custo real combinado menor ou igual ao somatório dos custos amortizados.

Para isso, definimos uma função potencial, normalmente denotada por Φ, que associa o estado atual da estrutura (ou coleção de estruturas) a um número real. Precisamos que Φ satisfaça as seguintes propriedades:

- $\Phi(S_0) = 0$, sendo S_0 o estado inicial;
- $\Phi(S_i) \geq 0$, para qualquer estado S_i.

Intuitivamente, podemos pensar na função potencial como a "energia" armazenada pelo estado atual da estrutura, de forma que quando uma operação custosa é feita, a energia é liberada. Se conseguimos, então, limitar o potencial da estrutura em qualquer momento do tempo, podemos limitar o custo total das operações.

Isso é formalizado da seguinte forma. Seja O_i a operação que leva do estado S_{i-1} para o estado S_i. Seja $R(O_i)$ o custo real da operação O_i. Definimos o custo amortizado $A(O_i)$ de uma operação:

$$A(O_i) = R(O_i) + \Big(\Phi(S_i) - \Phi(S_{i-1})\Big),$$

ou seja, o custo amortizado é o custo real mais a diferença de potencial que a operação causa. Definir o custo amortizado dessa forma é útil, pois para qualquer sequência de q operações $O_1, O_2, \ldots, O_q$, podemos computar o custo amortizado total $\mathcal{A}$, e ver que isso limita superiormente o custo total $\mathcal{R}$.

$$\mathcal{A} = \sum_{i=1}^{q} A(O_i) = \sum_{i=1}^{q} \Big(R(O_i) + \Big(\Phi(S_i) - \Phi(S_{i-1})\Big)\Big)$$

$$= \sum_{i=1}^{q} \big(R(O_i)\big) + \Phi(S_q) - \Phi(S_0) = \sum_{i=1}^{q} \big(R(O_1)\big) + \Phi(S_q) \geq \sum_{i=1}^{q} R(O_i) = \mathcal{R}$$

$$\therefore \mathcal{A} \geq \mathcal{R}$$

Isso significa que faz sentido dizer que esse é o custo amortizado das operações, pois o custo real total de qualquer sequência de operações não é maior que o custo amortizado total da sequência de operações, mesmo que uma operação individualmente possa ter custo real maior que seu custo amortizado.

Essa não é a única forma de definir complexidade amortizada, mas ela é bastante útil porque o potencial depende apenas do estado atual, e não da sequência de operações que foram feitas. Além disso, uma vantagem dessa análise é que podemos ter várias operações diferentes, e a análise pode revelar complexidades diferentes para cada operação.

Teorema 1. *Existe uma estrutura de dados capaz de realizar as operações* INSERT*,* DELETE*,* SEARCH *e* SPLIT *em tempo* $\mathcal{O}(\log U)$ *amortizado e pior caso, e a operação* MERGE *em tempo* $\mathcal{O}(1)$ *amortizado.*

Demonstração. Seja a função potencial Φ da coleção de *tries* igual ao somatório do número de nós em cada *trie*. As operações SEARCH e DELETE não criam nós, portanto o custo amortizado de ambas é no máximo seu custo real. As operações INSERT e SPLIT criam no máximo $\mathcal{O}(\log U)$ nós, portanto seu custo amortizado é $\mathcal{O}(\log U) + \mathcal{O}(\log U) = \mathcal{O}(\log U)$. Por fim, a operação MERGE, pelo Algoritmo 1, tem custo real proporcional a quantidade de nós apagados, que é exatamente a quantidade em que o potencial diminui ao fazer a operação. Portanto, seu custo amortizado é $\mathcal{O}(1)$. □

1.5. Otimização de Memória

Da forma como descrevemos, a estrutura de dados gasta $\mathcal{O}(n \log U)$ de memória para representar n elementos em $[0, U]$. Entretanto, é possível reduzir o consumo de memória para $\mathcal{O}(n)$: basta notar que, como a *trie* contém n folhas, podemos comprimir os caminhos longos da *trie*, ou seja, não precisamos representar os nós internos diferentes da raiz que só possuem um filho (ver Figura 1.4).

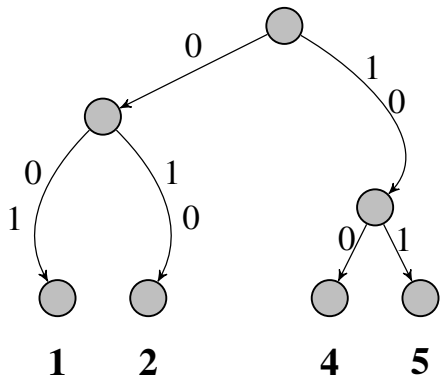


Figura 1.4. A *trie* comprimida da Figura 1.2.

Com isso, só precisamos representar no máximo $2n$ nós, se tivermos n folhas, e obtemos $\mathcal{O}(n)$ de memória. As operações podem ser facilmente ajustadas para essa representação compacta.

1.6. Outras Operações

É possível ampliar a estrutura para suportar outras operações. Se guardarmos, em cada nó da *trie*, o número de folhas presentes na sub-árvore relativa ao nó, conseguimos fazer operações referentes à quantidade de elementos:

7. SEARCH_K(S, k): retorna o k-ésimo menor elemento de S. Se S possui menos que k elementos, retorna -1.
8. SPLIT_K(S, k): retorna um par de conjuntos (A, B), tal que $A \cup B = S$, $A \cap B = \emptyset$, $|A| = k$ e $a \leq b \ \forall a \in A, b \in B$. O conjunto S é destruído.
9. COUNT(S, x): retorna o número de elementos de S que são menores que x.

A operação 7 é bastante simples. Se o número de folhas na raiz é menor que k, retornamos -1. Caso contrário, começando pela raiz da *trie*, repetimos o seguinte processo até chegarmos a uma folha: se na esquerda temos uma quantidade de folhas maior ou igual a k, descemos para a esquerda. Caso contrário, subtraímos de k a quantidade de folhas presentes na esquerda, e descemos para a direita. A folha em que terminamos é o valor buscado. Como só descemos por um caminho, a complexidade é $\mathcal{O}(\log U)$.

Para fazer a operação 8, fazemos um processo similar. Começando pela raiz, copiamos todos os nós que visitamos para uma nova *trie* A. Se na esquerda temos menos que k folhas, descemos para a esquerda. Caso contrário, movemos a sub-árvore da esquerda para A, subtraímos de k a quantidade de folhas na sub-árvore do nó movido e descemos para a direita. Atribuímos a B os nós restantes na *trie*. Novamente, como descemos por um caminho apenas, o custo da operação é $\mathcal{O}(\log U)$.

A operação 9, por fim, é implementada da seguinte forma. A partir da raiz, descemos seguindo a representação binária de x. Cada vez que descemos para a direita, incrementamos a resposta na quantidade de folhas da sub-árvore do filho da esquerda. Ao chegar a uma folha ou um nó NULL, retornamos o valor corrente.

Operação	Amortizado	Pior Caso
MAKESET	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(\log U)$	$\mathcal{O}(\log U)$
DELETE	$\mathcal{O}(\log U)$	$\mathcal{O}(\log U)$
SEARCH	$\mathcal{O}(\log U)$	$\mathcal{O}(\log U)$
SPLIT	$\mathcal{O}(\log U)$	$\mathcal{O}(\log U)$
MERGE	$\mathcal{O}(1)$	$\mathcal{O}(n \log U)$
SEARCH_K	$\mathcal{O}(\log U)$	$\mathcal{O}(\log U)$
SPLIT_K	$\mathcal{O}(\log U)$	$\mathcal{O}(\log U)$
COUNT	$\mathcal{O}(\log U)$	$\mathcal{O}(\log U)$

Tabela 1.1. Complexidades das Operações.

É fácil verificar que essas operações funcionam bem com nossa função potencial, pois as operações 7 e 9 não criam nós, portanto seu custo amortizado é no máximo seu

custo real. A operação 8 só cria no máximo $\mathcal{O}(\log U)$ novos nós, então seu custo amortizado é $\mathcal{O}(\log U)$. Na Tabela 1.1 temos as complexidades de todas as operações. Outro detalhe é que podemos ampliar facilmente a estrutura para lidar com multiplicidade de elementos; para isso basta armazenar a multiplicidade nas folhas.

1.7. Resultados Experimentais

A estruturas de dados que resolve o problema *Mergeable Dictionary* foi implementada como descrita, na linguagem `C++` [1]. Notamos que a estrutura de *trie* tem performance comparável a ABBs, como ilustrado no Gráfico 1.1 (entradas em negrito são as estruturas descritas neste trabalho).

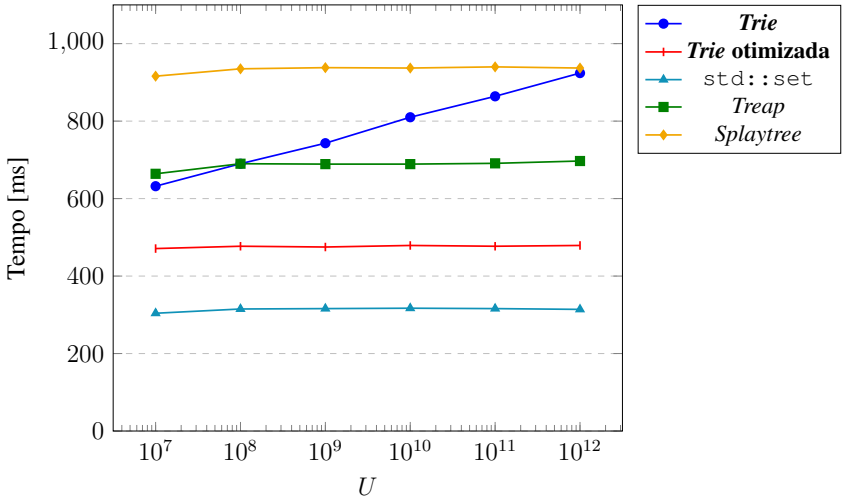


Gráfico 1.1. Tempo médio de 10 rodadas para executar 10^6 operações aleatórias de INSERT, DELETE e SEARCH, variando U.

Comparamos a estrutura com `std::set`, nativa do `C++`, que é implementada usando uma *Red-Black Tree* [7], que efetua as operações em $\mathcal{O}(\log n)$ no pior caso [8]; *Treap*, uma árvore binária de busca randomizada que tem altura $\mathcal{O}(\log n)$ com alta probabilidade [9]; *Splaytree*, cujas operações têm custo $\mathcal{O}(\log n)$ amortizado [3].

A *trie* otimizada é a estrutura com a otimização de memória. Notamos, pelo Gráfico 1.1, que a estrutura de *trie* aparenta crescer linearmente no gráfico, que está em escala logarítmica, assim exibindo a complexidade $\mathcal{O}(\log U)$ das operações.

Porém, a estrutura de *trie* otimizada tem a mesma complexidade, mas exibiu um padrão constante em função de U. Acreditamos que isso se deu pelo fato das operações serem aleatórias, então a altura da árvore comprimida é próxima de $\log n$, e também é possível que a alocação e o acesso a memória sejam o gargalo do algoritmo.

Para testar o comportamento das operações SPLIT e MERGE, implementamos o MERGE utilizando o algoritmo *segment merge*, que faz a operação em $\mathcal{O}(\log n \log U)$ amortizado [4]. Mas, ao medir os tempos vimos que o algoritmo *segment merge* se mostrou muito eficiente para operações aleatórias, como pode ser visto no Gráfico 1.2. Além disso, não foi possível notar dependência em U, para operações geradas aleatoriamente.

[1] `https://github.com/brunomaletta/DynamicArray`

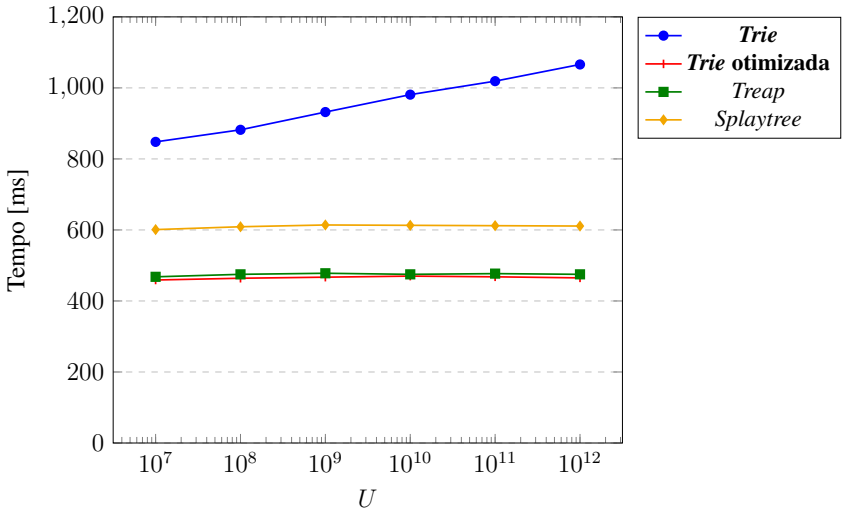


Gráfico 1.2. Tempo médio de 10 rodadas para executar 10^6 operações aleatórias de INSERT, DELETE, SEARCH, SPLIT e MERGE, variando U.

Para tentar forçar o pior caso do algoritmo *segment merge*, implementamos uma classe de instâncias descrita em [10], que, para $n = 2^{2k}$, fazem com que uma ABB que faz SPLIT e JOIN em $\Omega(\log n)$ leve tempo $\Omega(n \log^2 n)$ para fazer $\Theta(n)$ operações de SPLIT e MERGE usando o algoritmo *segment merge*.

As estruturas de *trie* levam tempo $\Theta(n \log n)$ para essas instâncias, e em [10] é dito que essa classe de instâncias para *Splaytrees* leva tempo $\mathcal{O}(n \log n)$. Isso é consistente com nossos experimentos. Observamos que a *Treap* leva cerca de 4 vezes o tempo da estrutura de *trie*, mas medir o tempo se mostrou intratável pois o tamanho das instâncias cresce exponencialmente, então os tempos ficam rapidamente incomparáveis.

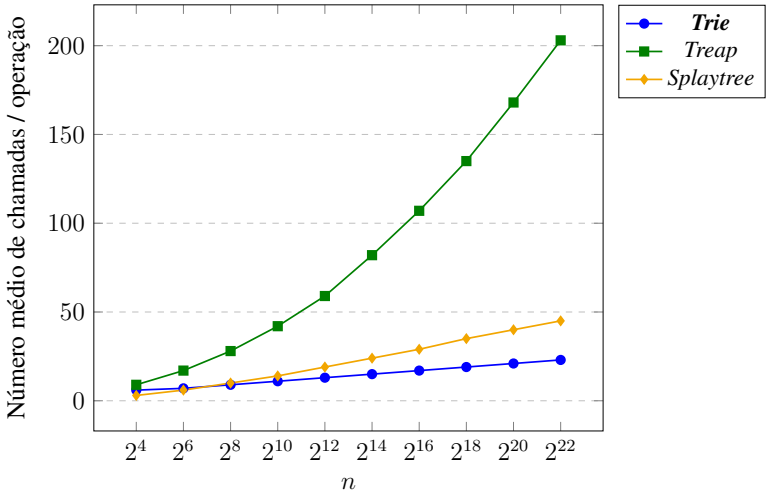


Gráfico 1.3. Numero médio de chamadas de 10 rodadas para executar $\mathcal{O}(n)$ operações de SPLIT e MERGE sobre n chaves em $[0, n)$, variando n.

Por esse motivo, em vez do tempo medimos o número médio de chamadas de função por operação para realizar as operações, e observamos o comportamento esperado das estruturas (ver Gráfico 1.3). Note que o eixo x está em escala logarítmica, então uma

estrutura que faz em média $\Theta(\log n)$ chamadas por operação devem se comportar como uma reta, e uma que faz $\Theta(\log^2 n)$ deve se comportar como uma parábola.

1.8. Conclusão e Trabalhos Futuros

Neste capítulo foi estudado a representação de conjuntos de inteiros utilizando uma *trie*, que permite que façamos operações de SPLIT e MERGE de forma eficiente. Utilizando de análise de complexidade por função potencial, mostramos como obter tempo $\mathcal{O}(\log U)$ amortizado para todas as operações, se estamos representando subconjuntos de $\{0, 1, 2, 3, \ldots, U\}$.

A estruturas de dados foi implementadas em C++, e o código está disponível de forma *open source* [2]. Comparamos o tempo de execução de nossa implementação com implementações de outras ABBs, e vemos que a estrutura de *trie* tem eficiência comparável a ABBs para valores de $U \leq 10^{12}$. Com isso vemos que é viável o uso da estrutura em aplicações reais, uma vez que ela permite que façamos uma gama bem maior de operações, sem perda significativa de eficiência.

Um problema em aberto é a possibilidade fazer resolver o problema *Mergeable Dictionary* em tempo amortizado $\mathcal{O}(\log n)$ ou, ainda, no pior caso, e se *Splaytrees* satisfazem esse limite usando o algoritmo *segment merge* descrito em [4].

[2] https://github.com/brunomaletta/DynamicArray

Capítulo 2

Representação de *Arrays*

2.1. Introdução

O problema de ordenação é dos problemas mais clássicos na Computação. Formalmente, dado um *array* (vamos assumir que seja um *array* de inteiros, por simplicidade), desejamos permutar seus valores de tal forma que cada elemento seja maior ou igual ao elemento anterior. Esse problema é de grande interesse porque é mais fácil trabalhar com dados ordenados: podemos fazer, por exemplo, busca em tempo logarítmico e acessar o k-ésimo menor elemento de forma trivial em tempo constante.

Esse problema é bastante estudado, e é conhecido que um algoritmo de ordenação por comparação precisa executar $\Omega(n \log n)$ comparações para ordenar um *array* de n elementos [1]. Existem muitos algoritmos que atingem esse limite inferior, e são, portanto, assintoticamente ótimos [11, 12].

É interessante também manter a propriedade de ordenação e suportar alterações nos dados, como adição e remoção de itens. Isso é feito tradicionalmente com árvores binárias de busca, que nos permite fazer operações de inserção, remoção e busca em tempo logarítmico na quantidade de itens armazenados [2, 3].

Outra situação de interesse é quando não queremos representar os dados ordenados, mas sim mantê-los em uma outra ordem. Por exemplo, se tivermos um *array* a tal que a_i nos fala o valor de uma ação no dia i, podemos perguntar quais foram os k maiores valores da ação do dia i ao dia j, ou seja, quais são os k maiores valores do *array* considerando os índices de i a j. Esse problema é bastante estudado e é conhecido como *range reporting* [13].

Neste capítulo será estudado uma variação da operação de *range reporting*, que chamamos de PARTITION, que, dado *array* e um inteiro k, queremos separar, em um novo *array* ordenado, os k menores valores do *array*. Isso se difere por não estarmos interessados em obter uma cópia dos k menores valores, mas sim em removê-los da estrutura, em uma nova instância dela. Isso nos dá a esperança de fazer a operação em tempo sub-linear, ou seja, $o(n)$.

2.2. Trabalhos Relacionados

O problema de, dado um *array* e um inteiro k, descobrir qual é o k-ésimo menor valor do *array* é chamado de *selection* ou *k-th order statiscic*. Isso pode ser feito acessando-se a k-ésima posição do *array* ordenado, o que leva tempo $\Theta(n \log n)$ no pior caso para um *array* de tamanho n ($\Theta(n \log n)$ para ordenar, $\mathcal{O}(1)$ para acessar a k-ésima posição). Porém, em 1973, foi mostrado que é possível resolver o problema em tempo linear no pior caso com um algoritmo recursivo [14].

Outro problema de interesse é o problema chamado *partial sorting*, que se trata de, dado um *array* de tamanho n e um inteiro k, obter uma cópia ordenada dos k menores valores do *array*. É possível resolver o problema em $\Theta(n \log n)$ simplesmente ordenando o *array* inteiro. Mas, em vez disso, podemos achar o k-ésimo menor elemento, separar os k menores e em seguida ordená-los, em $\mathcal{O}(n + k \log k)$. Como consequência do limite inferior para ordenação [1], esse algoritmo tem complexidade ótima.

O problema *range partial sorting* consiste em responder várias consultas de *partial sorting* para determinados intervalos do *array* definidos por um par de índices. Ou seja, dados inteiros i, j, k, tal que $1 \leq i \leq j \leq n$ e $1 \leq k \leq j - i + 1$, queremos uma lista ordenada dos k menores valores entre as posições de i até j do *array*. Em [15] foi mostrado uma estrutura de dados que, após construção em tempo $\mathcal{O}(n \log \log n)$, responde tais consultas em $\mathcal{O}(\log n + k \log \log n)$. Outra estrutura de dados, apresentada em [16], consegue responder as consultas em tempo $\mathcal{O}(\log \log n + k)$, após uma construção em tempo $\mathcal{O}(n \log n)$. É possível, ainda, responder as consultas em tempo ótimo, $\mathcal{O}(k)$, usando também $\mathcal{O}(n \log n)$ de pré-processamento [17]. Uma revisão bibliográfica dessas estruturas pode ser vista em [13].

Entretanto, essas estruturas de dados lidam com o problema estático, ou seja, o *array* não sofre alterações entre as consultas. Um tipo interessante de alteração são as do tipo SPLIT, que significa quebrar um *array* em dois a partir de um índice (de forma que um dos *arrays* contenha os valores dos x menores índices, o outro *array* contenha os demais valores, e o *array* original seja destruído) e CONCATENATE, que se trata de concatenar dois *arrays*. Desejamos, assim, manter uma coleção de *arrays* sujeitas às operações de SPLIT e CONCATENATE, e também a consultas de *partial sorting*. Note que com essas operações é possível também responder consultas de *range partial sorting*.

Uma solução possível para isso seria manter os *arrays* representados por árvores binárias de busca implícitas, como descrito em [12], o que permitiria fazer o SPLIT e CONCATENATE em $\mathcal{O}(\log n)$. Para o *partial sorting*, podemos sequencialmente encontrar e copiar o mínimo em tempo logarítmico k vezes, o que levaria tempo $\mathcal{O}(k \log n)$ [18]. Em [18], porém, foi provado que existe uma estrutura de dados capaz de responder consultas de *partial sorting* em $\mathcal{O}(\log^*_\varphi(n) k \log(k))$ enquanto suportam SPLIT e CONCATENATE em $\mathcal{O}(\log(n) \cdot \log^2(\log(n)))$, tal que $\log^*_\varphi(n)$ significa o logaritmo iterado de n na base $\varphi = \frac{1+\sqrt{5}}{2}$.

Podemos notar que, como a saída da consulta de *partial sorting* tem tamanho k e não altera a estrutura dos *arrays* representados, é necessário, pelo menos, um fator de k no tempo para se realizar essa operação. No pior caso, em que $k \in \Theta(n)$, isso pode ser bastante custoso.

É por esse motivo que estamos interessados em uma operação que, em vez de obter

uma cópia dos k menores valores, remova os k menores valores em uma nova estrutura. Como não criamos k novos itens por operação, temos esperança de fazer a operação em tempo amortizado $o(n)$, ou seja, sub-linear.

2.3. Um Novo Tipo Abstrato de Dados: BLOCK-SORTED ARRAY

Propomos um novo tipo abstrato de dados para representar *arrays* dinâmicos suportando operações de ordenação, que chamamos de BLOCK-SORTED ARRAY. A ideia é representar o *array* como uma concatenação de blocos ordenados (ver Figura 2.1).

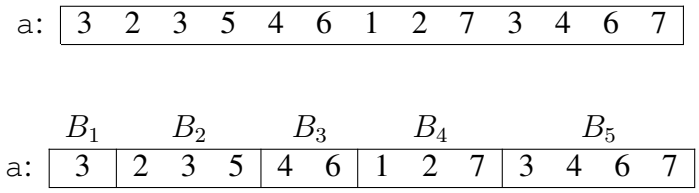


Figura 2.1. Exemplo de uma partição de blocos de um BLOCK-SORTED ARRAY. Note que cada bloco ($B_1, B_2, \ldots, B_5$) do *array* a está ordenado.

A representação de um *array* como um BLOCK-SORTED ARRAY não é única; por exemplo, separar cada valor em um bloco é sempre uma representação válida. Representar *arrays* dessa forma é útil, pois, caso queiramos ordenar um *array*, isso equivale a fazer o MERGE de todos os blocos.

Para implementar o tipo abstrato de dados, vamos representar cada bloco utilizando a estrutura de *trie* que descrevemos na Seção 1.3, assim conseguiremos fazer as operações descritas, em especial MERGE e SPLIT_K.

Queremos ser capazes de concatenar *arrays*, assim como quebrar um *array* em uma dada posição. Por esse motivo, vamos implementar o tipo abstrato de dados usando uma árvore binária de busca (ABB) balanceada implícita, e em cada nó armazenamos uma estrutura de *trie*, de forma que uma travessia em ordem (isto é, visitando a sub-árvore da esquerda, a chave do nó atual e depois a sub-árvore da direita) dê os blocos na ordem do *array* representado (ver Figura 2.2).

Ao fazer isso, podemos usar operações canônicas de ABB, como SPLIT, ou seja, quebrar uma ABB em duas, uma contendo os k menores elementos, e a outra contendo o restante dos elementos, e JOIN, que une duas ABBs em uma, se os elementos de uma são todos menores que os elementos da outra. Isso pode ser feito em $\mathcal{O}(\log n)$ no pior caso, para n elementos na ABB [19].

Os itens armazenados na ABB são *tries*, mas nunca comparamos duas *tries*: as chaves são a posição relativa dos blocos no *array*, e podemos computar a posição de cada bloco ao descer na ABB, se guardarmos os tamanhos das sub-árvores. Por esse motivo que chamamos a ABB de **implícita**, pois as chaves são, na verdade, o índice de cada bloco, porém isso não é armazenado explicitamente, mas sim computado quando necessário.

Usando as operações de JOIN e SPLIT em ABBs, conseguimos, respectivamente, concatenar dois *arrays* e quebrar um *array* em alguma fronteira de seus blocos. Mas não estamos limitados às fronteiras dos blocos, pois sabemos que podemos fazer um SPLIT

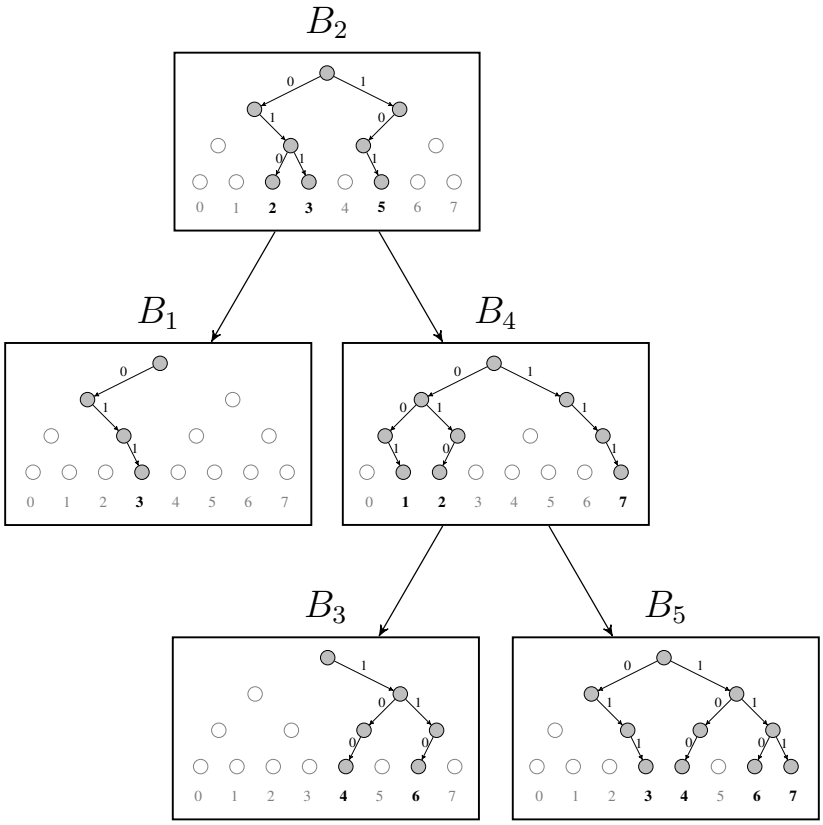


Figura 2.2. Implementação de um BLOCK-SORTED ARRAY sobre o *array* a (Figura 2.1) como ABB implícita que armazena *tries* representando os blocos.

também na estrutura de dados referente a um bloco. Assim, conseguimos quebrar o *array* em qualquer posição que desejarmos.

Vamos listar o que desejamos poder fazer com o BLOCK-SORTED ARRAY. Para isso definimos o problema *Range-Sorting Dynamic Array*, em que mantemos coleção dinâmica de *arrays* com valores em $\{0, 1, 2, 3, \ldots, U\}$, sujeita às seguintes operações:

(i). MAKEARRAY(): retorna um *array* vazio.
(ii). SEARCH(v, k): retorna o valor da k-ésima posição do *array* v.
(iii). CONCATENATE(a, b): retorna um *array* que é a concatenação dos *arrays* a e b. Os *arrays* a e b são destruídos.
(iv). SPLIT(v, k): retorna um par de *arrays* (a, b), tal que $|a| = k$ e a concatenação de a e b é v.
(v). REVERSE(v): reverte o *array* v, de forma que o primeiro e último valor são trocados entre si, assim como o segundo e penúltimo, etc.
(vi). SORT(v): ordena o *array* v de forma não-decrescente.

Podemos usar a estrutura BLOCK-SORTED ARRAY para resolver o problema. A operação (i) é trivial. A operação (ii) se baseia em descer na ABB, utilizando os tamanhos das sub-árvores. Descemos por um caminho até encontrar a *trie* em que o valor buscado se encontra, e efetuamos uma operação de SEARCH_K na *trie*, e assim obtemos o valor desejado. Descer na ABB tem custo $\mathcal{O}(\log n)$, assumindo que n é o tamanho do *array*, e, consequentemente, um limite superior para o número de blocos. Como precisamos ainda fazer uma operação SEARCH_K, o custo da operação (ii) é $\mathcal{O}(\log n + \log U)$ no pior caso.

A operação (iii) pode ser feita como foi previamente descrito: basta efetuar uma operação de JOIN nas ABBs, como vemos em [19]. O custo da operação é $\mathcal{O}(\log n)$ no pior caso. A operação (iv) pode ser feita com um SPLIT na ABB e no máximo um SPLIT_K, portanto podemos efetuá-la em $\mathcal{O}(\log n + \log U)$.

Para a operação (v), precisamos armazenar duas *flags* em cada nó da ABB, uma armazenando se precisamos reverter a sub-árvore relativa ao nó, e outra representando se a *trie* na verdade está revertida em relação à representação no *array*. Essa primeira *flag* pode ser propagada apenas quando necessário de forma preguiçosa (*lazy*), e isso não interfere na complexidade das outras operações. Portanto, precisamos apenas alterar uma *flag*, então fazemos a operação em tempo $\mathcal{O}(1)$ no pior caso.

Por fim, a operação (vi) pode ser feita de forma bastante simples: basta efetuar MERGE em todas as *tries*, ficando com apenas um bloco contendo todos os elementos do *array*. O pior caso da operação é $\Theta(n \log U)$, se iterarmos por todos as *tries*. Porém, como provaremos no Teorema 2, o custo amortizado da operação (vi) é $\mathcal{O}(1)$.

Teorema 2. *Existe uma estrutura de dados capaz de realizar as operações* MAKEARRAY *e* REVERSE *em tempo* $\mathcal{O}(1)$ *amortizado e pior caso,* CONCATENATE *em tempo* $\mathcal{O}(\log n)$ *amortizado e pior caso,* SEARCH *e* SPLIT *em tempo* $\mathcal{O}(\log n + \log U)$ *amortizado e pior caso e a operação* SORT *em tempo* $\mathcal{O}(1)$ *amortizado.*

Demonstração. Seja a função potencial Φ da coleção de *arrays* igual ao somatório do número de nós em cada *trie*. A operação MAKEARRAY cria uma quantidade constante de nós em uma *trie*, portanto seu custo amortizado e real são ambos $\mathcal{O}(1)$. As operações REVERSE, CONCATENATE e SEARCH não criam nós em nenhuma *trie*, portanto seu custo amortizado é no máximo seu custo real, então a operação REVERSE tem custo amortizado $\mathcal{O}(1)$, a operação CONCATENATE tem custo amortizado $\mathcal{O}(\log n)$ e a operação SEARCH tem custo amortizado $\mathcal{O}(\log n + \log U)$.

A operação SPLIT faz um SPLIT na ABB, que não cria nós em *tries*, e depois faz um SPLIT_K em no máximo uma *trie*. O SPLIT_K cria no máximo $\mathcal{O}(\log U)$ nós, portanto o custo amortizado da operação é $\mathcal{O}(\log n + \log U) + \mathcal{O}(\log U) = \mathcal{O}(\log n + \log U)$.

Por fim, a operação SORT tem custo real proporcional a quantidade de nós de *tries* apagados, pelo Algoritmo 1, uma vez que toda *trie* compartilha pelo menos a raiz com qualquer outra *trie*. Portanto, seu custo real é proporcional à redução do potencial que a operação proporciona, então o custo amortizado da operação é $\mathcal{O}(1)$. □

Operação	Amortizado	Pior Caso
MAKEARRAY	$\mathcal{O}(1)$	$\mathcal{O}(1)$
SEARCH	$\mathcal{O}(\log n + \log U)$	$\mathcal{O}(\log n + \log U)$
CONCATENATE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
SPLIT	$\mathcal{O}(\log n + \log U)$	$\mathcal{O}(\log n + \log U)$
REVERSE	$\mathcal{O}(1)$	$\mathcal{O}(1)$
SORT	$\mathcal{O}(1)$	$\mathcal{O}(n \log U)$

Tabela 2.1. Complexidades das Operações do BLOCK-SORTED ARRAY.

As complexidades amortizadas e pior caso das operações podem ser vistas sucintamente na Tabela 2.1, assumindo que a coleção possui n elementos no total que todos os elementos pertencem ao intervalo $[0, U]$.

2.4. Outras Operações

Agora trataremos da operação PARTITION que definimos. Ou seja, queremos manter nossa colação de *arrays*, suportando as operações já descritas, e também a operação de, dado um *array* e um inteiro k, separar do *array* seus k menores valores em um novo *array* ordenado. Por simplicidade, assumimos que não há valores repetidos nos *arrays*.

Inicialmente vamos nos atentar para uma variação dessa operação, que chamaremos de PARTITIONVALUE. Essa operação recebe um *array* e um valor x, e desejamos separar do *array* todos os valores menores que x, em um *array* ordenado. Em seguida temos as definições formais das duas operações.

(vii). PARTITIONVALUE(v, k): retorna um *array* com os k menores valores de v ordenados. Esses valores são removidos de v.
(viii). PARTITION(v, x) retorna um *array* com todos os valores de v que são menores que x. Esses valores são removidos de v.

2.4.1. Implementação das Operações

Vamos agora descrever como implementar as operações (vii) e (viii) usando nossa implementação de BLOCK-SORTED ARRAY. Propomos duas implementações para a operação (vii). A primeira delas se dá da seguinte forma.

Queremos separar do *array* todos os valores menores que x. Achamos, então, o primeiro valor maior ou igual a x, e chamamos a operação SPLIT para separar o prefixo até essa posição, pois queremos remover esses valores. Em seguida, encontramos o primeiro valor menor que x, e fazemos outro SPLIT até essa posição.

Repetimos esse processo, quebrando o *array* em pedaços em que todos os valores são menores do que x e pedaços em que todos os valores são maiores ou iguais a x, alternadamente. Concatenamos, usando a operação CONCATENATE, os pedaços com valores maiores ou iguais a x. O resultado dessa concatenação é o que sobra do *array* inicial. Concatenamos da mesma forma os pedaços menores que x em um novo *array*. Por fim, ordenamos esse novo *array* usando a operação SORT, e o retornamos.

Note que é simples encontrar o primeiro valor menor que x ou maior ou igual a x, guardando o máximo e mínimo das sub-árvores da estrutura (tanto da ABB quanto das *tries*). O algoritmo para a operação (vii) pode ser visto então no Algoritmo 2.

Agora descreveremos uma segunda implementação para a operação (vii). O que podemos fazer é, quanto existir um valor no *array* que seja menor que x (que podemos verificar armazenando o mínimo), descemos na ABB até encontrar a *trie* que contém o mínimo do *array*. Fazemos então um SPLIT na *trie* que contém o mínimo, removendo todos os valores menores que x da *trie*.

Ao fim desse processo, unimos todas as *tries* removidas usando a operação MERGE, e isso nos dá o *array* que devemos retornar. A implementação desse algoritmo pode ser visto no Algoritmo 3. Definimos a função SPLITFROMSMALLESTTRIE(v, x),

Algoritmo 2 PARTITIONVALUE(v, x)

```
w ← MAKEARRAY()
temp ← MAKEARRAY()
enquanto v for não vazio faça
    pref1 ← primeira posição de v que não é menor que x
    (v⁻, v) ← SPLIT(v, pref1)
    w = CONCATENATE(w, v⁻)

    pref2 ← primeira posição de v que é menor que x
    (v⁻, v) ← SPLIT(v, pref2)
    temp ← CONCATENATE(temp, v⁻)
v ← temp
SORT(w)
retorne w
```

que recebe um *array* v e um valor x. Seja T a *trie* da representação de v que contém o valor mínimo em v. A função remove de T todos os valores menores que x, e retorna esses valores removidos.

Algoritmo 3 PARTITIONVALUE(v, x)

```
w ← MAKEARRAY()
enquanto o velor mínimo em v for menor que x faça
    T ← SPLITFROMSMALLESTTRIE(v, x)
    temp ← array representado por T
    w ← CONCATENATE(w, temp)
SORT(w)
retorne w
```

Finalmente, para a PARTITION(v, k), podemos fazer busca binária para descobrir o x tal que a função PARTITIONVALUE(v, x) (operação (vii)) fará o que desejamos. Se, no meio do *loop* principal da implementação da operação (vii) o *array* que será retornado ficar com mais que k elementos, paramos a operação (vii) e devemos então buscar um x menor. Se, por outro lado, o *array* retornado pela operação (vii) tiver menos que k elementos, devemos buscar um x maior.

Fica claro, portanto, que a operação (viii) pode ser implementada com um *overhead* multiplicativo de $\mathcal{O}(\log U)$ em relação à operação (vii).

2.4.2. Análise do Custo das Operações

Seja $P(n, U)$ o número amortizado de vezes que o *loop* do Algoritmo 2 executa para uma coleção de *arrays* de tamanho total n que representa valores até U. Seja $Q(n, U)$ o número amortizado de vezes que o *loop* do Algoritmo 3 executa sob as mesmas condições. É fácil ver que $P(n, U) \leq Q(n, U)$, pois em cada iteração do Algoritmo 2 o conjunto de itens adicionados ao *array* que será retornado é um superconjunto de um conjunto de itens que será adicionado ao *array* retornado em alguma iteração do Algoritmo 3.

Tentamos provar um limite sub-linear para $P(n, U)$, mas falhamos. Acreditamos, entretanto, que $P(n, u)$ tenha crescimento logarítmico em função de n e possivelmente de U (Conjectura 1).

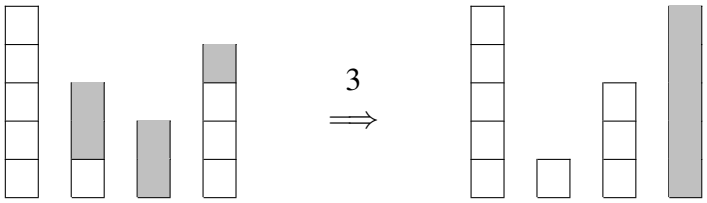
Conjectura 1. *Assumindo $U = n$, temos que $P(n, U) \in \mathcal{O}(\log n)$*

Descreveremos agora uma tentativa interessante de provar um limite sub-linear para $P(n, U)$. Vamos analisar $Q(n, U)$, que sabemos que limita $P(n, U)$ superiormente. Podemos abstrair o Algoritmo 3 e pensar que ele escolhe alguns blocos, remove alguns elementos de alguns dos blocos selecionados e une os elementos removidos em um novo bloco. O custo $Q(n, U)$ é a quantidade amortizada de blocos selecionados pelo algoritmo.

Vamos definir o que chamaremos de *Problema das Pilhas de Moedas* (PPM). Podemos ver que um limite sub-quadrático para o PPM implica um limite sub-linear para $Q(n, U)$, e, consequentemente, para $P(n, U)$, que é o que desejamos. Para ver isso, podemos interpretar as pilhas como sendo os blocos, e cada movimento é uma iteração da *loop* no Algoritmo 3.

Problema das Pilhas de Moedas

Existem tem n moedas, inicialmente cada uma em uma pilha de tamanho 1. No movimento i, são selecionadas k_i pilhas, e retira-se uma quantidade não nula de moedas de cada pilha selecionada (é possível remover uma pilha por completo). As moedas removidas são então empilhadas em uma nova pilha. O **custo** do movimento é k_i. Qual é o **custo máximo** para se fazer n operações?



Exemplo de um movimento de custo 3.

É claro que o PPM é $\mathcal{O}(n^2)$, mas acreditamos que exista um limite mais justo. Tentamos provar isso, mas não conseguimos. Provamos, entretanto, um limite mais justo para uma versão do PPM em que em cada movimento podemos remover apenas uma moeda de cada pilha selecionada. No Teorema 3 mostramos esse resultado. Chamamos essa versão do problema de *Problema das Pilhas de Moedas Restrito* (PPMR).

Teorema 3. *O Problema das Pilhas de Moedas Restrito é $\Theta(n\sqrt{n})$.*

Demonstração. Vamos definir um grafo G com n vértices, em que cada vértice representa uma moeda. Dois vértices i e j de G terão uma aresta se as moedas que i e j representam estão na mesma pilha.

Inicialmente, então, o grafo não possui arestas. No movimento i, se escolhemos k_i pilhas (e, consequentemente, temos custo k_i), vemos que no máximo $n - 1$ arestas são removidas do grafo (para cada pilha selecionada removemos seu tamanho menos um), e adicionamos $\binom{k_i}{2}$ arestas. Ao final dos n movimentos, temos no máximo $\binom{n}{2}$ arestas. Então

$$\sum_{i=1}^{n}\left(\binom{k_i}{2}-n\right)\leq\binom{n}{2}$$
$$\sum_{i=1}^{n}\binom{k_i}{2}\leq\frac{n^2-n}{2}+n^2=\frac{3n^2-n}{2}$$

Como estamos interessados apenas no crescimento assintótico, podemos escrever que, para alguma constante $c\in\mathbb{R}$,

$$\sum_{i=1}^{n}k_i^2\leq cn^2,$$

mas

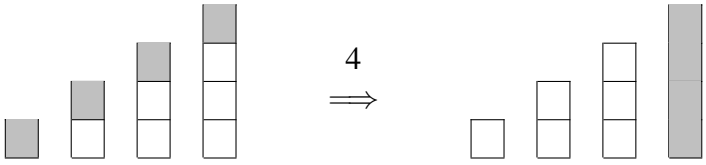
$$\frac{1}{n}\left(\sum_{i=1}^{n}k_i\right)^2\leq\sum_{i=1}^{n}k_i^2,$$

que é um caso particular da desigualdade das somas de Chebyshev [20]. Por fim, temos que

$$\frac{1}{n}\left(\sum_{i=1}^{n}k_i\right)^2\leq\sum_{i=1}^{n}k_i^2\leq cn^2$$
$$\left(\sum_{i=1}^{n}k_i\right)^2\leq cn^3$$
$$\sum_{i=1}^{n}k_i\leq\sqrt{c}\cdot n^{\frac{3}{2}},$$

ou seja, o PPMR é $\mathcal{O}(n\sqrt{n})$. Para ver que o PPMR é $\Omega(n\sqrt{n})$, considere a seguinte construção. Vamos assumir, sem perda de generalidade, que n é um número triangular, ou seja, que $n=1+2+\cdots+r$.

Agora as operações se darão da seguinte forma: sempre pegamos uma moeda de cada uma das r maiores pilhas. Isso é possível, porque inicialmente temos uma moeda por pilha, depois teremos uma pilha de tamanho r e outras de tamanho 1, depois uma de tamanho r, uma de tamanho $r-1$, e outras de tamanho 1, etc., até que tenhamos uma de tamanho r, uma de tamanho $r-1$, ..., uma de tamanho 1. A partir desse momento, o estado das pilhas não muda, e podemos repetir essa operação de custo r indefinidamente.



Exemplo de movimento que não muda o estado com $r=4$.

Vemos, por fim, como $r \in \Theta(\sqrt{n})$, que o limite de $\mathcal{O}(n\sqrt{n})$ é assintoticamente justo, o que completa a prova. □

Acreditamos que esse resultado ainda seja verdade mesmo para o problema não restrito, o PPM (ver Conjectura 2). Se esse for o caso, então isso implica que $Q(n, U) \in \mathcal{O}(\sqrt{n})$, e, por consequência, que $P(n, u) \in \mathcal{O}(\sqrt{n})$.

Conjectura 2. *O Problema das Pilhas de Moedas é* $\Theta(n\sqrt{n})$.

Para verificar essa hipótese, executamos um código força-bruta que verifica todas as possibilidades do PPM, buscando maximizar o custo. Os resultados podem ser vistos no Gráfico 2.1, e parecem corroborar a Conjectura 2.

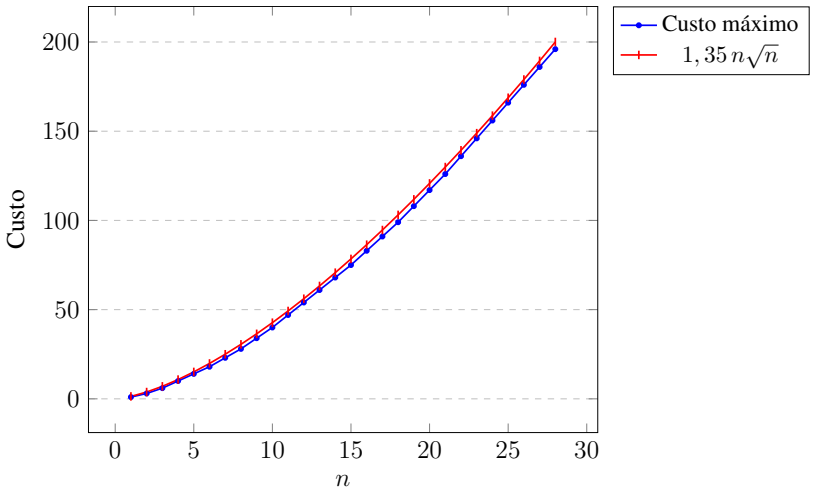


Gráfico 2.1. Custo máximo do PPM variando n, e uma função $\Theta(n\sqrt{n})$.

Por fim, exibimos uma tabela com as complexidades de tempo amortizado e pior caso para todas as operações descritas do BLOCK-SORTED ARRAY na Tabela 2.2.

Operação	Amortizado	Pior Caso
MAKEARRAY	$\mathcal{O}(1)$	$\mathcal{O}(1)$
SEARCH	$\mathcal{O}(\log n + \log U)$	$\mathcal{O}(\log n + \log U)$
CONCATENATE	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
SPLIT	$\mathcal{O}(\log n + \log U)$	$\mathcal{O}(\log n + \log U)$
REVERSE	$\mathcal{O}(1)$	$\mathcal{O}(1)$
SORT	$\mathcal{O}(1)$	$\mathcal{O}(n \log U)$
PARTITIONVALUE	$\mathcal{O}(P(n, U)(\log n + \log U))$	$\mathcal{O}(n(\log n + \log U))$
PARTITION	$\mathcal{O}(P(n, U)(\log n + \log U) \log U)$	$\mathcal{O}(n(\log n + \log U) \log U)$

Tabela 2.2. Complexidades de Todas as Operações do BLOCK-SORTED ARRAY.

2.5. Conclusão e Trabalhos Futuros

Neste capítulo foi introduzido um novo tipo abstrato de dados chamado BLOCK-SORTED ARRAY para representar uma coleção dinâmica de *arrays*. Provamos que uma implementação do BLOCK-SORTED ARRAY utilizando a estrutura de *trie* vista no

Capítulo 1 nos permite manter uma colação de *arrays* sujeita às operações de quebrar um *array* em dois em uma posição, concatenar *array* e ordenar *arrays* de forma eficiente, com complexidade logarítmica amortizada.

Além disso, estudamos uma operação que definimos, chamada de PARTITION, que se trata de remover, de forma ordenada, os k menores valores de um *array*. Mostramos uma implementação dessa operação usando no BLOCK-SORTED ARRAY, porém não conseguimos provar um limite sub-linear amortizado para a operação, embora conjecturemos que ele exista.

Definimos um problema bastante interessante, o *Problema das Pilhas de Moedas*. Resolvemos um caso particular do problema, e conjecturamos que o problema geral tem o mesmo limite superior que o caso particular que estudamos. Esse limite implicaria em um limite amortizado sub-linear para a operação de PARTITION. Fizemos experimentos empíricos de força-bruta, que corroboram nossas conjecturas.

Como trabalhos futuros, pretendemos tentar provar nossas conjecturas, especialmente um limite superior sub-linear amortizado para a operação PARTITION.

Referências

[1] Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest e Clifford Stein: *Introduction to algorithms*. MIT press, 2009.

[2] Adel'son-Vel'skii, George M e Evgenii Mikhailovich Landis: *An algorithm for organization of information*. Em *Doklady Akademii Nauk*, volume 146, páginas 263–266. Russian Academy of Sciences, 1962.

[3] Sleator, Daniel Dominic e Robert Endre Tarjan: *Self-adjusting binary search trees*. Journal of the ACM (JACM), 32(3):652–686, 1985.

[4] Farach, Martin e Mikkel Thorup: *String matching in lempel—ziv compressed strings*. Algorithmica, 20(4):388–404, 1998.

[5] Iacono, John e Özgür Özkan: *Mergeable dictionaries*. Em *International Colloquium on Automata, Languages, and Programming*, páginas 164–175. Springer, 2010.

[6] Karczmarz, Adam: *A simple mergeable dictionary*. Em *15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[7] *GCC - Set implementation*. `https://gcc.gnu.org/onlinedocs/gcc-4.6.2/libstdc++/api/a01064_source.html`. Data de acesso: 23/03/2021.

[8] Guibas, Leo J e Robert Sedgewick: *A dichromatic framework for balanced trees*. Em *19th Annual Symposium on Foundations of Computer Science (SFCS 1978)*, páginas 8–21. IEEE, 1978.

[9] Seidel, Raimund e Cecilia R Aragon: *Randomized search trees*. Algorithmica, 16(4):464–497, 1996.

[10] Rutschmann, Daniel: *Worst case segmented merge instance for treaps*. `https://codeforces.com/blog/entry/67980?#comment-522890`. Data de acesso: 23/03/2021.

[11] Williams, John William Joseph: *Algorithm 232: heapsort*. Commun. ACM, 7:347–348, 1964.

[12] Knuth, Donald E: *The art of computer programming: sorting and searching*. Em *The art of computer programming: sorting and searching*, páginas 723–723. Addison-Wesley, 1975.

[13] Skala, Matthew: *Array range queries*. Em *Space-Efficient Data Structures, Streams, and Algorithms*, páginas 333–350. Springer, 2013.

[14] Blum, Manuel, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, Robert Endre Tarjan *et al.*: *Time bounds for selection*. J. Comput. Syst. Sci., 7(4):448–461, 1973.

[15] He, Meng, J Ian Munro e Gelin Zhou: *Path queries in weighted trees*. Em *International Symposium on Algorithms and Computation*, páginas 140–149. Springer, 2011.

[16] Nekrich, Yakov e Gonzalo Navarro: *Sorted range reporting*. Em *Scandinavian Workshop on Algorithm Theory*, páginas 271–282. Springer, 2012.

[17] Brodal, Gerth Stølting, Rolf Fagerberg, Mark Greve e Alejandro López-Ortiz: *Online sorted range reporting*. Em *International Symposium on Algorithms and Computation*, páginas 173–182. Springer, 2009.

[18] Liu, Jiamou e Kostya Ross: *Dynamic Partial Sorting*. arXiv preprint arXiv:1402.2712, 2014.

[19] Tarjan, Robert Endre: *Data structures and network algorithms*. SIAM, 1983.

[20] Wikipedia: *Chebyshev's sum inequality*. `https://en.wikipedia.org/wiki/Chebyshev%27s_sum_inequality`. Data de acesso: 08/09/2021.