Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Algoritmos e Estruturas de Dados Aleatorizadas

Pedro Teotonio de Sousa

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Orientador: Prof. José Coelho de Pina Júnior

São Paulo
10 de março de 2021

Algoritmos e Estruturas de Dados Aleatorizadas

Pedro Teotonio de Sousa

Esta é a versão original da monografia
elaborada pelo candidato Pedro
Teotonio de Sousa, tal como
submetida à Comissão Julgadora.

Resumo

Pedro Teotonio de Sousa. **Algoritmos e Estruturas de Dados Aleatorizadas**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Aleatorização, por meio sequências de números pseudoaleatórios, é utilizada na prática para manter certas propriedades em estruturas de dados, auxiliar em coleta de amostras, em aplicações visuais, jogos e muitas outras coisas. Esse texto apresenta como essas sequências são geradas na prática, assim como alguns algoritmos e estruturas de dados que os utilizam, muitas vezes melhorando sua complexidade de tempo e espaço e simplificando seu código.

Palavras-chave: aleatorização, treap, skip list, filtro de Bloom

Abstract

Pedro Teotonio de Sousa. **Algoritmos e Estruturas de Dados Aleatorizadas**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

Randomization, by means of pseudorandom number sequences, is used in practice to preserve certain properties on data structures, assist on sample collection, in visual applications, games and many others areas. This work shows how these sequences are generated in practice, as well as some algorithms and data structures that leverage them, often resulting in better time and space complexity and simplifying code.

Keywords: randomization, treap, skip list, filtro de Bloom

Sumário

1 Introdução **1**

2 Gerando números aleatórios **3**

2.1 Geradores congruentes lineares 3

2.2 Mersenne twister 5

2.3 Funções de hash 5

2.4 Testes de aleatoriedade 5

3 Entrada adversária **7**

3.1 Árvores de busca binária 7

3.2 Quicksort 10

3.3 Fecho convexo incremental 10

3.4 Ordenação por inserção 11

4 Treap **13**

4.1 Estrutura 13

4.2 Operações 14

4.3 Representando listas 18

4.4 Lazy propagation 19

5 Skip list **21**

5.1 Estrutura 21

5.2 Operações 22

5.3 Complexidade 25

5.4 Acesso aleatório 26

5.5 Utilizações 27

6 Filtro de Bloom **29**

6.1 Estrutura 29

6.2 Operações 30
6.3 Encontrando valores ótimos 31
6.4 Construindo funções de hash 33
6.5 Utilização de memória 33
6.6 Adicionando mais elementos que o esperado 34
6.7 Remoção de elementos 34
6.8 Paralelização 35
6.9 Utilizações 35

7 Comentários finais **37**

Referências **39**

Capítulo 1

Introdução

Imprevisibilidade é um conceito útil em diversas áreas, como criptografia, simulações, jogos, amostragem e nos algoritmos e estruturas de dados que serão apresentados nesse texto.

Dados são utilizados desde a pré-história (Carlisle, 2009), demonstrando que nossa utilidade para imprevisibilidade não é algo novo. Além disso, também existem exemplos mais modernos dos mais diversos de utilização de imprevisibilidade fora de computação. Um lixadeira roto-orbital, por exemplo, realiza movimentos aleatorizados para evitar marcas regulares em madeira.

https://www.finehomebuilding.com/2016/01/06/whats-the-difference-fine-finish-sanders-orbital-vs-random-orbit

Figura 1.1: *Diferença entre o padrão gerado por uma lixadeira orbital a esquerda e uma roto-orbital a direita.*

Para as aplicações que apresentaremos nesse texto, o que precisamos não é da imprevisibilidade em si, e sim coisas previsíveis que podemos extrair da imprevisibilidade.

Mesmo um pouco contraintuitivo, imprevisibilidade pode sim gerar coisas previsíveis. Por exemplo, se jogamos um dado de 6 lados milhares de vezes, podemos esperar que cada face ocorra em aproximadamente 1/6 do número total de jogadas. Essas previsibilidades emergentes são o que nos permite tornar algoritmos mais efetivos e manter propriedades de estruturas com baixo custo.

Este texto descreve um pouco mais sobre como computadores mimetizam imprevisibilidade e sobre como podemos aplicá-la em algoritmos e estruturas de dados para simplificar as operações.

No capítulo 2, trataremos da geração de sequências de números aleatórios e o que significa uma sequência de números aleatória para uma aplicação específica. Em seguida, no capítulo 3, vemos como a aleatorização da entrada de um algoritmo pode trabalhar a nosso favor, produzindo entradas bem comportadas que geram bom desempenho.

Um dicionário armazena pares (chave, valor) e implementa operações de busca, inserção e remoção. Procurar dados em uma coleção é uma operação muito comum, tornando o dicionário uma estrutura de dados muito utilizada. Apresentamos duas estruturas que implementam um dicionário e utilizam aleatorização para realizar suas operações de forma eficiente: uma árvore balanceada chamada **Treap** no capítulo 4, e a **Skip List**, implementada através de um conjunto de listas ligadas, no capítulo 5.

Finalmente, no capítulo 6, tratamos do **filtro de Bloom**, que utilizando funções de hash, consegue verificar se um elemento está em um conjunto com uma taxa de erro customizável, utilizando uma quantidade significativamente menor de memória que a soma dos elementos presentes.

Capítulo 2

Gerando números aleatórios

Um gerador de números aleatórios é, efetivamente, um gerador de imprevisibilidade. Queremos gerar sequências de números que se comportem como uma sequência imprevisível.

No contexto de computação, temos necessidade de grandes quantidades de números aleatórios, quantidades que uma pessoa com um dado, mesmo sendo tecnicamente um gerador de números aleatórios, não consegue suprir.

Nossa solução para esse problema no século 20 foi criar tabelas com números escolhidos de diversas formas, aleatórios o suficiente para as aplicações da época. (Tippett e Pearson, 1959) (*A million random digits: with 100.000 normal deviates* 1955) Hoje em dia conseguimos gerar milhares de números aleatórios em um piscar de olhos, e temos muitos testes estatísticos para avaliar a qualidade dessas sequências.

Existem diversos métodos para a geração de números aleatórios, sites como random.org, por exemplo, utilizam ruído atmosférico para gerar números aleatórios (Haahr, s.d.), entre outros projetos que utilizam movimentações em lâmpadas de lava, decaimento radioativo, entre outros.

Mesmo gerando boas sequências aleatórias, esses métodos são pouco práticos em algumas aplicações no nosso dia a dia. O que queremos é um método computacional relativamente simples e rápido para gerar uma sequência aleatória.

Mas não precisamos de uma sequência realmente aleatória, apenas algo que seja suficientemente aleatório para nossa aplicação.

Chamamos essa sequências de **pseudoaleatórias**.

Existem diversos métodos para geração de números pseudoaleatórios, descreveremos um pouco mais dois desses métodos a seguir.

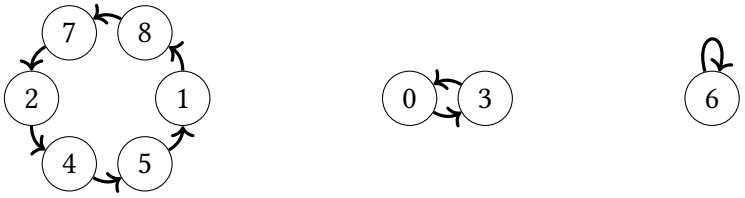
2.1 Geradores congruentes lineares

Um gerador congruente linear é uma função recorrente da seguinte forma:

$$X_{n+1} = (aX_n + c) \bmod m,$$

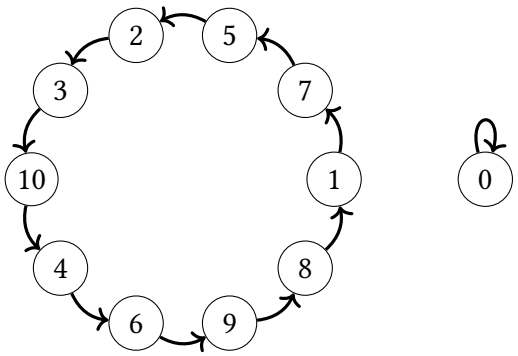
onde $X_0, X_1, \ldots$ é nossa sequência de valores pseudoaleatórios, e m, a e c são números inteiros, parâmetros da nossa recorrência. Chamamos m de **módulo**, a de **multiplicador**, c de **incremento**. Quando os demais parâmetros são escolhidos, o primeiro valor da sequência, X_0 define unicamente o restante da sequência, e o chamamos de **semente**.

Para $a = 5$, $c = 3$ e $m = 9$, por exemplo, temos o seguinte caso, onde os arcos apontam para o próximo elemento da sequência:



Pode parecer estranho que uma função cíclica tão simples gere valores satisfatoriamente aleatórios, e de fato, a gritante maioria das escolhas desses valores gera sequências bem distantes de serem aleatórias.

Boas escolhas para um gerador congruente linear envolvem escolhas de módulo, incremento e multiplicador para que essa sequência seja grande e passe em testes de aleatoriedade. Um bom começo para escolha desses valores pode ser ser a tentativa de gerar um sequência de tamanho máximo. Para isso podemos deixar $c = 0$ e escolher m primo e a como uma raiz primitiva módulo m. Para $c = 0$, $m = 11$ e $a = 7$, por exemplo, temos a seguinte sequência.



Note que isso é apenas um jeito de maximizar o tamanho do ciclo, não necessariamente gerando boas sequências aleatórias. Por exemplo, qualquer sequência com $a = c = 1$ gera ciclos de tamanho máximo, mas a sequência está longe de ser considerada aleatória para qualquer aplicação prática.

Outra escolha comum de parâmetros envolve escolher m como uma potência de 2 e $c = 0$, deixando o calculo da sequência bem eficiente mas gerando alguns problemas, como período máximo $\frac{m}{4}$ e bits menos significativos possuindo períodos menores que bits mais significativos.

2.2 Mersenne twister

O **Mersenne Twister** (Matsumoto e Nishimura, 1998) é um dos algoritmos de geração de números pseudoaleatórios mais utilizados. Assim como geradores congruentes lineares, ele também é baseados em uma recorrência, porém bem mais complexa.

Mesmo sendo muito mais sofisticado que um gerador congruente linear, ele ainda se baseia em uma recorrência simples para gerar uma sequência determinística de números que se comportam como uma sequência aleatória.

O Mersenne Twister tem esse nome pois seu período é um primo de Mersenne, um primo da forma $2^n - 1$. A versão mais comum utiliza um recorrência com período $2^{19937} - 1$, um número de 6002 dígitos decimais.

Além de um período gigantesco, o mt19937 garante equidistribuição de ordem 623, ou seja, sequências de até 623 números consecutivos da saída são uniformemente distribuídas, além de ser rápido e consumir poucos recursos.

Também existem múltiplas variantes que foram desenvolvidas para usos específicos, como o **CryptMT**, uma versão criptograficamente segura e o **TinyMT**, que possui período muito menor ($2^{127} - 1$) mas que utiliza menos recursos.

2.3 Funções de hash

Uma função de hash é qualquer função que tem dados de tamanho arbitrário como entrada e de tamanho fixo como saída.

Por ser uma função, uma mesma entrada sempre gera uma mesma saída, o que parece não combinar nem um pouco com o conceito de geração de aleatoriedade. O importante nesse caso é lembrar que, no geral, não estamos procurando algo realmente aleatório, e sim algo suficientemente aleatório.

No caso de funções de hash, as usadas na prática possuem algumas propriedades adicionais, como uniformidade. Ou seja, é esperado que toda saída seja tão frequente quanto qualquer outra para o conjunto de entradas possíveis.

Aqui já começamos a ver algumas semelhanças com sequências aleatórias, o que nos permite analisar funções de hash como se fossem sequências aleatórias, ao mesmo tempo que nos beneficiamos de serem funções determinísticas.

2.4 Testes de aleatoriedade

Um teste de aleatoriedade tenta quantificar a aleatoriedade de um certa sequência de números.

Por exemplo, como um teste muito básico, podemos determinar se cada número aparece uma quantidade similar de vezes em uma sequência. Se nossa sequência passar nesse teste não temos certeza que ela é aleatória, mas temos pelo menos uma evidência a favor dessa hipótese.

Isso é verdade no geral, dado a natureza de sequências aleatórias. Nenhum conjunto de teste consegue realmente determinar se uma sequência é aleatória, mas lembrando, não queremos uma sequência verdadeiramente aleatória, só precisamos que ela passe em um conjunto de testes forte o suficiente.

Na prática, existem diversos conjuntos de teste empíricos para sequências aleatórias, entre eles, Diehard tests (Marsaglia, 1995) e TestU01 (L'ecuyer e Simard, 2007).

Capítulo 3

Entrada adversária

Uma entrada que faz um algoritmo em particular apresentar desempenho pior que seu caso médio é chamada de **adversária**.

Esse tipo de entrada é bastante comum em competições de programação, testes de desempenho e também podem acontecer naturalmente.

Aleatorizar a entrada de um algoritmo com desempenho ruim para entradas adversárias gera um novo algoritmo que é, muitas vezes, surpreendentemente satisfatório.

A priori, parece que não mudamos absolutamente nada. Nosso conjunto de entradas é exatamente o mesmo, e muitas vezes o algoritmo em si não muda em absolutamente nada. Por consequência, nosso pior e melhor casos também são os mesmos.

O que aconteceu aqui é que nosso algoritmo tem seu desempenho atrelado a alguma propriedade da entrada. Se essa propriedade ruim é incomum no conjunto de todas as entradas, aleatorizar uma entrada específica gera uma entrada típica, que não terá essa propriedade ruim, melhorando o desempenho.

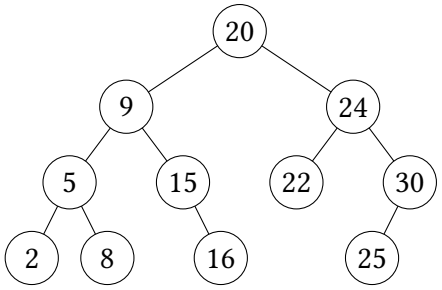
Existem alguns exemplos notórios do poder da aleatorização da entrada.

3.1 Árvores de busca binária

Uma **árvore de busca binária (BST)** é uma árvore enraizada onde cada nó tem um valor e dois filhos, um a esquerda e outro a direita. Uma invariante importante de uma BST é que para todo nó, todos os valores em sua subárvore esquerda são menores que o seu valor, e todos os valores da subárvore direita são maiores.

Temos como entrada uma lista de valores distintos e queremos construir uma BST com objetivo de minimizar os tempos de busca. A forma de buscar um valor na BST é começar na raiz da árvore e comparar o valor procurado com o valor do nó, decidindo se a busca irá continuar na subárvore esquerda ou direita.

Pela estrutura da árvore e a forma como a busca de um valor funciona, para minimizar o tempo de busca queremos minimizar a altura da árvore. O melhor que conseguimos fazer para uma árvore com n valores é uma altura $O(\log n)$. Conseguimos construir essa árvore



de uma maneira determinística: ordenamos os valores e escolhemos a mediana para ser a raiz da árvore, construindo a subárvore da esquerda e da direita com a mesma lógica, usando os valores restante.

Mesmo com um método determinístico simples para realizar essa tarefa, vamos utilizar um método aleatorizado que deixa a árvore com altura duas vezes maior em média. Por enquanto, esse método vai parecer apenas sub-ótimo, mas sua utilidade será mais clara quando falarmos de **Treap**.

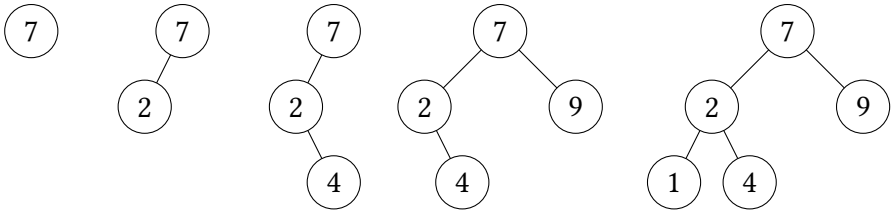
Primeiro vamos definir um algoritmo para construir a árvore: vamos inserir os elementos um por um de forma recursiva. Se a árvore estiver vazia, esse elemento vira a raiz, senão, vamos comparar os valores e decidir em qual das subárvores inserir.

```
struct BST {
    int value;
    BST* left, * right;
    BST(int v) {
        value = v;
        left = right = NULL;
    }
};

BST* insert(BST* root, int value) {
    if (root == NULL) {
        return new BST(value);
    }
    if (value < root->value) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }
    return root;
}
```

Para a lista $[7, 2, 4, 9, 1]$, por exemplo, temos os seguintes passos para a construção da árvore:

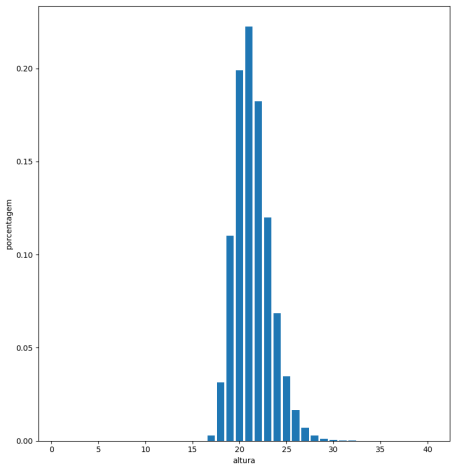
Claramente existem algumas entradas onde o nosso algoritmo cria uma árvore com altura igual a quantidade de elementos da lista, o que para uma árvore de busca é uma propriedade particularmente ruim.

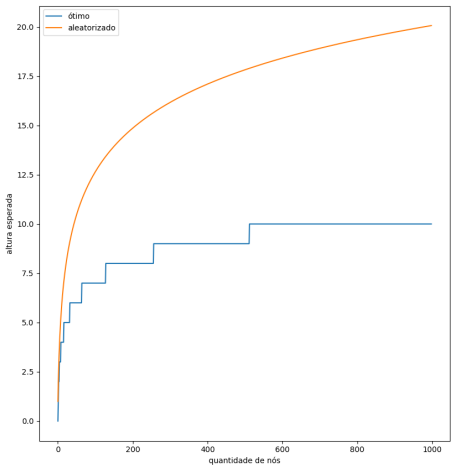


Essas entradas adversárias existem, mas são uma porcentagem muito pequena de todas as entradas possíveis. Para gerar um árvore com altura n, por exemplo, precisamos que o próximo elemento sempre seja o maior ou o menor dos elementos restantes, como em cada momento só existem 2 elementos com essa propriedade, a porcentagem das entradas que geram uma árvore de tamanho n é $\frac{2^n}{n!}$. Para $n = 10$ essa porcentagem já está na ordem de 10^{-4} e só melhora com o crescimento de n:

$$\lim_{n \to \infty} \frac{2^n}{n!} = 0$$

Claro que árvores de altura exatamente n não são nossa única preocupação, mas entradas ruins no geral são raras e podemos verificar isso experimentalmente: gerando 10^5 listas aleatórias de tamanho 10^3, observamos que não só entradas ruins são raras (nenhuma lista gerou uma árvore de altura maior que 33) quando geradas aleatoriamente, mas que o caso médio é particularmente próximo do ótimo para um algoritmo tão simples:





Conseguimos ver por aqui o que foi falado inicialmente. Entradas que geram uma árvore muito alta são tão incrivelmente raras que, para todos os efeitos práticos, temos um algoritmo $O(n \log n)$ gerando árvores de altura $O(\log n)$, mesmo quando o pior caso é um algoritmo $O(n^2)$ gerando árvores de altura $O(n)$.

3.2 Quicksort

Se no lugar de escolher um pivô aleatório escolhemos o primeiro valor da lista como pivô, o quicksort fica extremamente similar ao nosso algoritmo de geração de árvores binárias.

Na verdade, a quantidade de comparações para um elemento qualquer durante o quicksort é exatamente quantas comparações precisariam ser feitas para achar o lugar dele na árvore. Dessa forma, todas as observações feitas para a árvore de busca se aplicam a essa versão simples do quicksort, que com a aleatorização da entrada passa a ter tempo esperado $O(n \log n)$ mesmo com entradas adversárias.

3.3 Fecho convexo incremental

O fecho convexo de um conjunto de n pontos P em d dimensões é o menor politopo convexo d-dimensional que contém todos os pontos de P.

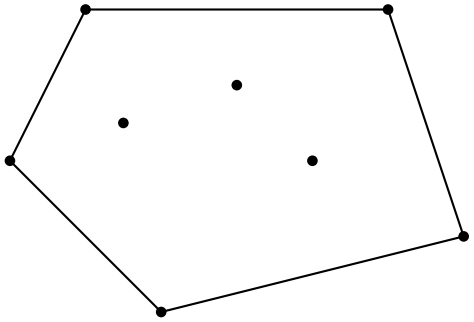


Figura 3.1: *Fecho convexo 2D.*

No fecho convexo incremental para duas dimensões, inserimos os pontos um a um, atualizando o fecho depois de toda inserção. Quando inserimos um ponto p em um fecho convexo H, se esse ponto está dentro de H, não fazemos nada. Se p está fora, vamos remover um conjunto conexo de arestas e adicionar exatamente duas. Aqui aparece um algoritmo simples, podemos percorrer todas as arestas de H quando vamos adicionar um ponto p, decidir quais remover e quais inserir e atualizar H.

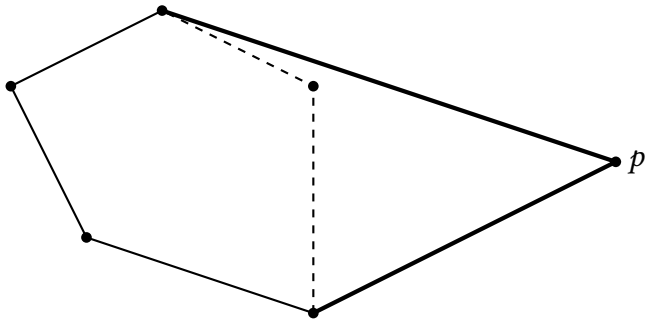


Figura 3.2: *Inserção do ponto p no fecho convexo, arestas pontilhadas serão removidas e as escuras serão adicionadas*

Isso é muito lento para o nosso objetivo de um algoritmo $O(\log n)$, precisamos descobrir mais rapidamente quais arestas remover. Como nossas arestas removidas formam um conjunto conexo, se soubermos de pelo menos uma aresta e_p que precisa ser removida, podemos olhar arestas adjacentes e iterar somente pelas arestas que precisam ser removidas, gerando um custo amortizado $O(1)$. Para conseguir realizar essa tarefa, mantemos as arestas do fecho convexo, em ordem horária ou anti-horária, em uma lista duplamente ligada.

Para descobrir a aresta e_p de cada ponto p, podemos escolher um ponto c dentro do fecho e descobrir qual aresta de H intersecta o raio $\vec{cr}$. Podemos fazer isso no início do algoritmo para descobrir e_p para todos os pontos, e atualizar essas arestas a medida que elas são removidas pela inserção de outros pontos. Isso ainda nos da uma complexidade $O(n^2)$ já que potencialmente precisamos atualizar e_p para todos os pontos restantes em todas as inserções.

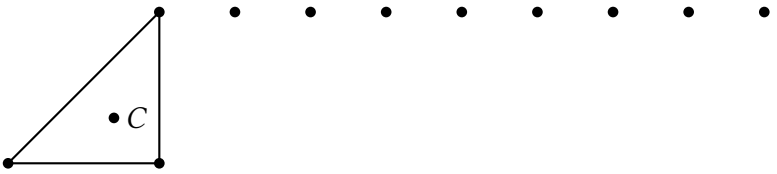


Figura 3.3: *Exemplo de um caso que o algoritmo não aleatorizado demora. Inserindo os pontos que não estão no fecho convexo da esquerda para a direita, sempre vamos precisar atualizar e_p para todos os pontos restantes.*

Mais uma vez, aleatorizar a entrada nos ajuda a diminuir a frequência dos casos que geram tempo quadrático, de forma que apenas aleatorizar a ordem de inserção de pontos gera um algoritmo com complexidade esperada $O(\log n)$. Além disso, a ideia geral do algoritmo incremental para o fecho convexo pode facilmente ser transferida para dimensões maiores, guardando faces para cada ponto, no lugar de arestas, e estruturas que conseguem iterar por um conjunto conexo de faces.

A técnica de aleatorizar a entrada e guardar um pouco mais de informação para conseguir resolver o problema com um código simples e eficiente aparace em muitos outros problemas geométricos (Clarkson, 1987) (Clarkson, 1988).

3.4 Ordenação por inserção

A ordenação por inserção nos da um bom exemplo das situações em que aleatorizar a entrada acaba não ajudando.

Como o algoritmo sempre mantém um prefixo ordenado, e faz isso pegando o último elemento e levando ele até a posição correta por meio de trocas com o elemento anterior, temos que a quantidade de trocas que serão feitas é exatamente a quantidade de inversões da lista original.

Acontece que existem mais jeitos de uma lista ter muitas inversões do que poucas. Na verdade, a quantidade esperada de inversões de uma permutação de tamanho n é $n(n-1)/4$, apenas duas vezes menos que o pior caso para o algoritmo, e muito pior que os melhores casos.

Capítulo 4

Treap

Como já vimos anteriormente, aleatorizar a ordem de inserção dos elementos em uma árvore de busca binária deixa a altura esperada dessa árvore $O(\log n)$.

Infelizmente, apenas aleatorizar a ordem de inserção não nos permite realizar atualizações em uma BST (como inserir ou remover elementos) e ainda manter sua altura pequena.

Uma **Treap** é basicamente uma BST que adiciona um valor a mais a cada nó chamado prioridade e mantém uma invariante em cima da estrutura da árvore e esse valor. Essa mudança é o suficiente pra habilitar atualizações em tempo $O(\log n)$ enquanto mantendo sua altura esperada também $O(\log n)$.

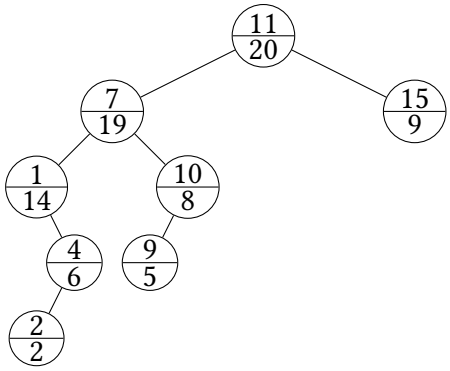
4.1 Estrutura

Assim como uma BST, a estrutura da treap é recursiva, cada nó da treap possui uma treap à esquerda e à direita.

Cada nó de uma treap tem uma **chave** e uma **prioridade**. Vamos representar um nó da treap com o seguinte símbolo, onde k é a chave do nó, e p é sua prioridade.

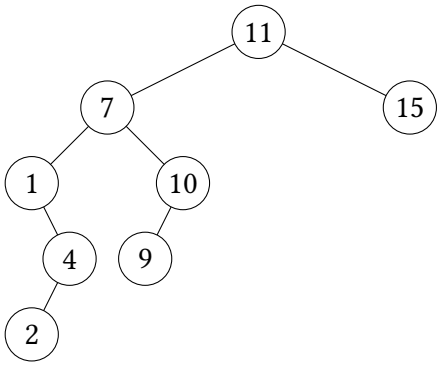
A chave é o que queremos guardar e buscar em nossa treap, já a prioridade é atribuída aleatoriamente a cada nó. Note que guardamos só a chave pois isso é suficiente para construir a estrutura, mas podemos guardar múltiplas coisas além da chave, tornando a treap um dicionário. A treap mantém uma árvore de busca binária em relação as chaves e um heap em relação as prioridades.

Em outras palavras, a chave de um nó é maior que a de seu filho à esquerda e menor que a de seu filho à direita, já a prioridade de um nó é maior que a prioridade de todos os seus filhos diretos e indiretos.



Essa estrutura é exatamente igual a de uma BST onde os elementos foram inseridos por ordem decrescente de prioridade.

Para a treap acima, note que inserir os valores em ordem de maior prioridade, $[11, 7, 1, 15, 10, 4, 9, 2]$, em uma BST, gera exatamente a mesma estrutura da árvore.



Como as prioridades foram geradas aleatoriamente, ordenar por prioridade é equivalente a permutar as chaves aleatoriamente, assim temos nossa altura $O(\log n)$.

Aqui usamos inteiros como chaves, mas qualquer tipo com o operador < definido também pode ser utilizado.

```
struct Treap {
    int key;
    int priority;
    Treap* left, * right;
    Treap(int k) {
        key = k;
        priority = rand();
        left = right = NULL;
    }
};
```

4.2 Operações

A maneira com que vamos definir as operações em uma treap será a partir de três operações base, `find`, `split` e `merge`. Realizar essas operações para realizar inserções e

remoções pode não ser a forma mais eficiente na prática, mas deixa a implementação muito mais simples e facilita a visualização de uma variação muito importante da treap.

Find

A operação `find` encontra um elemento com uma certa chave k. Como a treap tem as propriedades de uma árvore de busca binária (BST), essa operação é exatamente igual a uma BST.

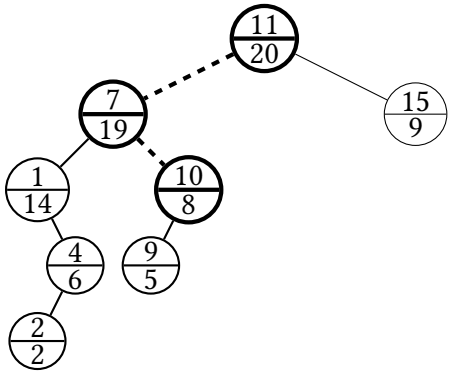


Figura 4.1: *O caminho percorrido por `find(10)` em nossa treap exemplo*

```
Treap* find(Treap* h, int k) {
    if (h == NULL) return NULL
    if (h->key < k) return find(h->right, k);
    if (k < h->key) return find(h->left, k);
    return h;
}
```

Split

A operação `split` recebe dois parâmetros: a raiz da treap e uma chave k. O resultado será duas treaps, onde uma delas possui todos os nós com chave menor que k e a outra possui todos os nós com chave maior ou igual a k.

Isso pode ser implementado de forma bem simples: se a chave da raiz da treap é menor que k, então chamamos `split` para a subtreap direita da raiz. Agora só precisamos reconstruir, trocando a subtreap direita pela subtreap esquerda resultante desse `split`. Fazemos algo simétrico para quando a raiz é maior ou igual a k.

```
pair<Treap*, Treap*> split(Treap* h, int k) {
    if (h == NULL) return {NULL, NULL};
    if (h->key < k) {
        auto split_right = split(h->right, k);
        h->right = split_right.first;
        return {h, split_right.second};
    } else {
        auto split_left = split(h->left, k);
        h->left = split_left.second;
```

```
        return {split_left.first, h};
    }
}
```

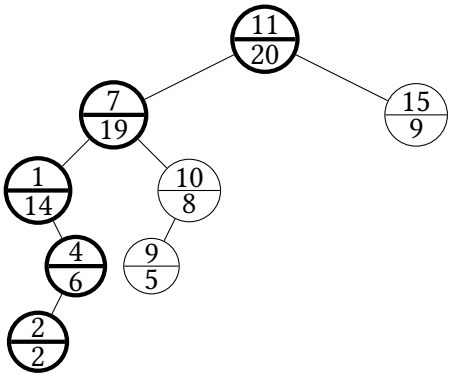


Figura 4.2: *Quando chamamos `split(3)` para a raiz da treap, acontecem chamadas recursivas de `split(3)` para todos os nós marcados*

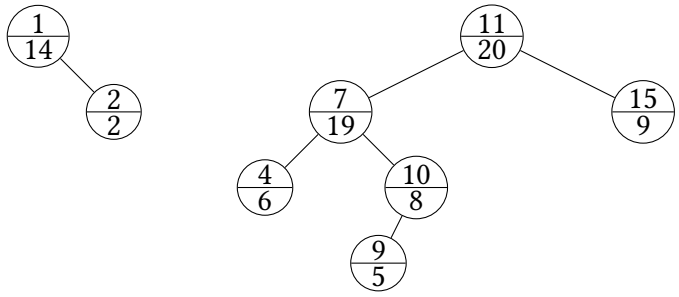


Figura 4.3: *Um `split(3)` em nossa treap original gera as seguintes árvores. Note como ambas ainda possuem todas as invariantes de uma treap e que as chaves da treap à esquerda são menores que todas as chaves à direita.*

Merge

A operação de `merge` recebe duas treaps que chamaremos de `left` e `right` como parâmetros, onde todas as chaves de `left` precisam ser menores que todas as chaves de `right`.

A implementação dessa operação continua muito simples: como queremos manter a propriedade de que elementos em alturas menores tem maior prioridade, vamos escolher a treap com maior prioridade de raiz para ser a nova raiz, e chamar `merge` recursivamente das subtreaps relevantes.

```
Treap* merge(Treap* left, Treap* right) {
    if (left == NULL) return right;
    if (right == NULL) return left;
    if (left->priority < right->priority) {
        right->left = merge(left, right->left);
        return right;
    } else {
```

```
        left->right = merge(left->right, right);
        return left;
    }
}
```

Um `merge` das subtreaps da figura 4.3 apenas gera a treap original antes do `split`, mas se mudarmos a prioridade do elemento com chave 7 de 19 para 10, notamos que o `merge` gera uma estrutura bem diferente, mas que ainda preserva as propriedades da treap.

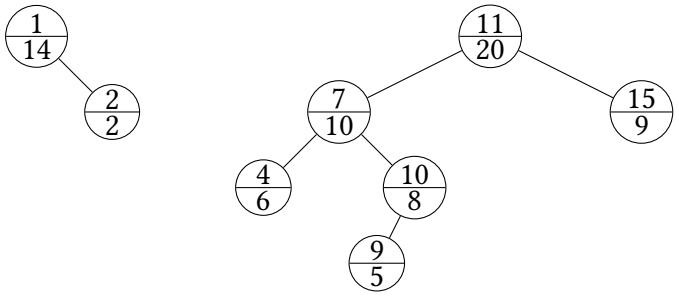


Figura 4.4: *Mesmas treaps da figura 4.3, apenas com a prioridade do elemento com chave* 7 *alterada de* 19 *para* 10

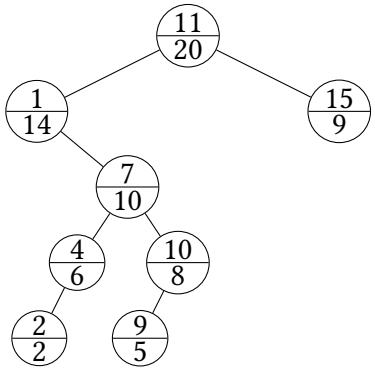


Figura 4.5: *Resultado do `merge` das treaps anteriores*

Inserção e remoção

Agora que definimos as operações de `split` e `merge` podemos utilizá-las para realizar operações de inserção e remoção com facilidade.

Para inserir um elemento com chave k, criamos o nó equivalente a esse elemento, fazemos um `split` em k, e depois dois `merges`, o primeiro da subtreap esquerda com k, e outra dessa treap resultante com a subtreap direita.

Para remover um elemento com chave k, basta achar esse elemento na treap e realizar um `merge` de suas subtreaps.

4.3 Representando listas

Treaps podem ser utilizadas para representar listas, o que permite a realização de operações em intervalos da lista.

Podemos usar a posição de um elemento na lista como sua chave na treap mas isso cria restrições nos tipos de operação que podemos fazer em nossa treap. Por exemplo, uma operação de fazer um shift circular em um intervalo mudaria a chave de múltiplos elementos e seria muito custosa.

Para conseguir realizar esse tipo de operação, precisamos de uma maneira alternativa de descobrir aonde uma certa posição da lista está na treap, para que ainda seja possível realizar as operações usuais na estrutura.

Para fazer isso podemos, no lugar de guardar uma chave, manter qual o tamanho da subtreap de cada nó e manter esse valor atualizado com todas as operações. Assim, para encontrar o nó que representa uma posição, só precisamos olhar para a subtreap esquerda e depois para a direita, comparando seus tamanhos com a posição que queremos encontrar.

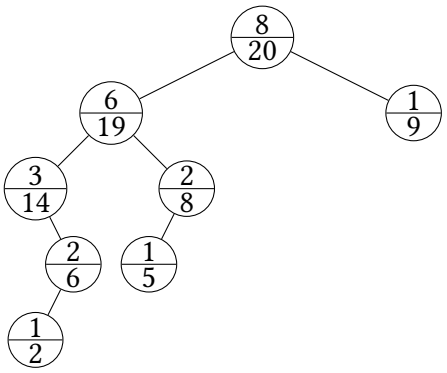


Figura 4.6: *Uma treap onde a chave foi substituída pela quantidade de nós da subtreap. Note que para que isso seja útil, precisamos guardar mais informações além da chave. Na representação de listas, podemos guardar o conteúdo daquela posição*

Note que a maioria das operações muda, não estamos comparando chaves, estamos comparando tamanhos de subtreaps. Para inserir um nó na posição p, fazemos um `split` de forma que a subtreap esquerda tenha tamanho p e a direita tenha tamanho $n - p$, e fazemos dois `merges`, com o elemento novo no meio, sempre lembrando de atualizar o tamanho da subtreap para todos os nós relevantes.

Podemos usar a mesma ideia de manter o tamanho da subtreap atualizado depois de cada operação para manter outras informações, como a soma, mínimo e máximo de subtreaps. Isso tem a consequência de que 2 `splits` para isolar um intervalo e uma consulta à raiz dessa treap terá essa informação apenas para esse intervalo, assim como também conseguimos atualizar esses valores para alguma posição da lista realizando dois `splits` que isolam essa posição, mudando seu valor e realizando `merges` para reconstruir a estrutura.

Combinando tudo isso, temos uma estrutura que consegue representar listas e fazer

operações bastante complexas que mudam a localização de múltiplos elementos da lista ao mesmo tempo, ainda mantendo a habilidade de atualizar uma posição e realizar perguntas em intervalos.

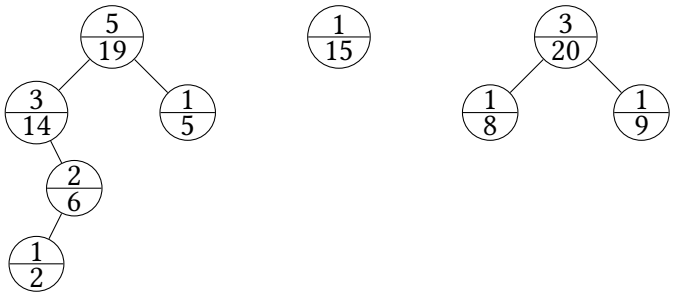


Figura 4.7: *Realizando um* `split` *para que a subtreap da esquerda tenha* 5 *nós. Note que os valores para os tamanhos da subtreap foram atualizados. O nó do meio é um novo elemento que será inserido na posição 5.*

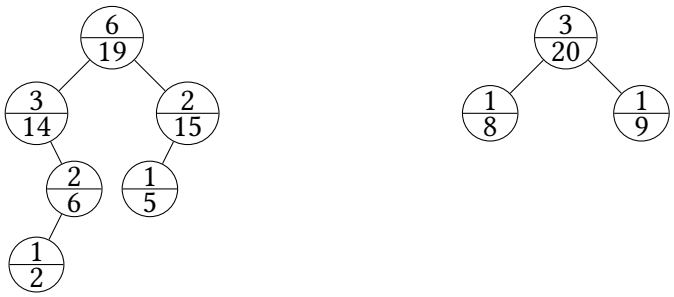


Figura 4.8: *Merge do novo nó com a subtreap esquerda.*

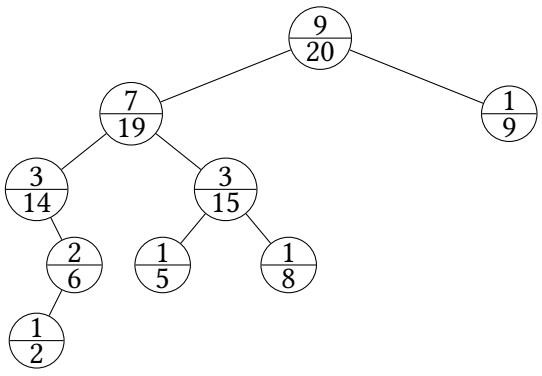


Figura 4.9: *Merge das duas subtreaps restantes, gerando uma treap que representa uma lista com um novo elemento na posição* 5.

4.4 Lazy propagation

Mesmo com tudo que a treap consegue fazer, ainda podemos utilizar lazy propagation para realizar atualizações em range e operações ainda mais complexas, mantendo a complexidade $O(\log n)$.

A ideia geral de lazy propagation em um intervalo é a mesma para as outras operações em listas. Quebramos a treap com 2 `splits` para isolar o intervalo e adicionamos alguma informação na raiz que informa que essa atualização precisa ser passado para os filhos.

No momento de visitar os filhos para realizar qualquer tipo de operação, esse valor é propagado, mantendo a propriedade de que sempre que isolamos um intervalo com `splits`, temos garantia que o valor presente na raiz dessa subtreap é o valor atualizado para todo o intervalo.

Capítulo 5

Skip list

A **Skip List** é uma alternativa de árvores balanceadas que usa um gerador de números aleatórios para manter sua estrutura, realizando busca, inserção e remoção em complexidade logarítmica. Sem precisar das operações complexas de rotação presentes nas implementações determinísticas de árvores balanceadas, a skip list tem uma constante muito baixa e uma implementação muito simples (Pugh, 1990).

5.1 Estrutura

A skip list é composta por vários níveis de listas ligadas ordenadas. O primeiro nível é uma lista ligada com todos os valores presentes na estrutura. Cada nível acima fica gradualmente mais esparso, com um elemento que aparece no nível i também aparecendo no nível $i + 1$ com probabilidade p. Valores de $p = \frac{1}{4}$ e $\frac{1}{2}$ são comumente usados.

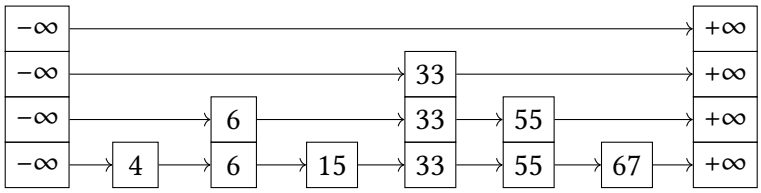


Figura 5.1: *Uma Skip List com 6 elementos. Note que o primeiro nível, representado aqui com os valores de baixo, possui todos os 6 valores da estrutura e a medida que subimos esses valores vão gradualmente desaparecendo.*

Cada uma dessas listas ligadas tem como primeiro elemento um **head** com valor $-\infty$ e como último elemento um nó com valor $+\infty$, ambos presentes em todos os níveis. Essa propriedade de níveis maiores com menos elementos é o que faz a skip list realizar suas operações rapidamente.

No código, representamos cada nó com uma chave e uma lista do próximo elemento da lista ligada para cada nível em que esse elemento está presente.

```
template <class T>
struct SkipListNode {
```

```
    T key;
    std::vector<SkipListNode<T>*> next;
};
```

O uso de $+\infty$ e $-\infty$ deixa as a estrutura mais consistente dado que estamos mantendo os valores ordenados, mas não é necessário na implementação. Aqui, representamos $-\infty$ com um nó qualquer e $+\infty$ com um nó nulo.

```
template <class T>
struct SkipList {
    SkipListNode<T>* head;
    SkipList() {
        head = new SkipListNode<T>;
        head->next.push_back(NULL);
    }
    std::vector<SkipListNode<T>*> lesser(T);
    bool find(T);
    void insert(T);
    void remove(T);
};
```

5.2 Operações

Antes de tratarmos das operações de inserção, busca e remoção, definimos uma operação `lesser` que facilitará o entendimento das outras operações.

Lesser

Essa operação encontra, para cada nível, o primeiro elemento na Skip List que é estritamente menor que uma dada chave k. Para fazer isso, a busca começa na **head**, no maior nível da estrutura. Em cada passo, comparamos k com o próximo elemento na lista ligada. Se k é maior, sabemos que o elemento atual não é o `lesser` desse nível e podemos dar um passo na lista ligada. Se k é menor ou igual, sabemos que esse elemento é o `lesser` desse nível, guardamos esse elemento e andamos para o nível abaixo. No final dessas operações, temos o `lesser` de k para cada nível da estrutura.

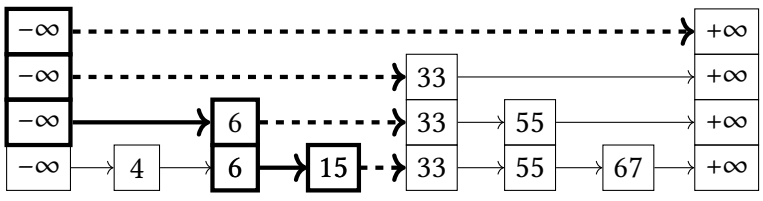


Figura 5.2: *`lesser(24)` aplicado na estrutura anterior. Os nós e arestas mais escuros foram visitados, e arestas pontilhadas saem dos valores escolhidos por `lesser` em cada nível. Note que todas as arestas pontilhadas saem de um valor menor que 24 para um maior.*

```
template <class T>
std::vector<SkipListNode<T>*> SkipList<T>::lesser(T key) {
```

```
    std::vector<SkipListNode<T>*> nodes;
    int current_level = head->next.size() - 1;
    SkipListNode<T>* current = head;
    while (current_level >= 0) {
        SkipListNode<T>* next_node = current->next[current_level];
        while (next_node != NULL && next_node->key < key) {
            current = next_node;
            next_node = current->next[current_level];
        }
        nodes.push_back(current);
        current_level--;
    }
    std::reverse(nodes.begin(), nodes.end());
    return nodes;
}
```

Busca

Se uma chave k está na estrutura, é garantido que o `lesser` do primeiro nível aponta diretamente para esse elemento. Só precisamos verificar se isso é verdade.

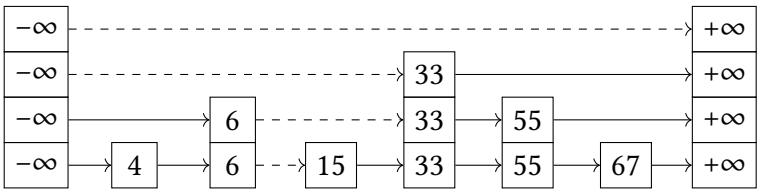


Figura 5.3: *Temos arestas pontilhadas saindo dos nós encontrados por `lesser(15)`. Note que a aresta pontilhada no primeiro nível aponta para um nó com valor 15, assim sabemos que 15 está na estrutura.*

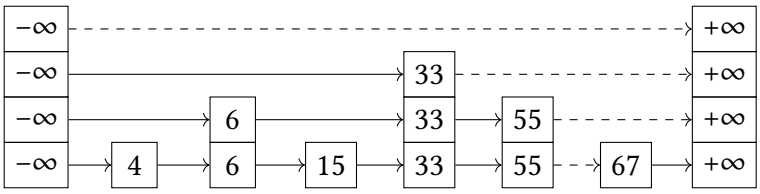


Figura 5.4: *Fazendo algo similar para `lesser(60)` notamos que a aresta pontilhada do primeiro nível aponta para 67, e não 60. Assim sabemos que 60 não está presente na estrutura.*

```
template <class T>
bool SkipList<T>::find(T key) {
    auto node = lesser(key)[0]->next[0];
    return node != NULL && node->key == key;
}
```

Inserção

Antes de inserir uma chave k, escolheremos quantos níveis esse elemento terá na estrutura. Como dito anteriormente, esse elemento sempre vai aparecer no primeiro nível,

a partir de agora com probabilidade p ele também vai aparecer no nível de cima. Assim, todo elemento inserido aparece em um prefixo dos níveis da estrutura.

Precisamos manter a propriedade de que cada nível é uma lista ligada ordenada. A função `lesser` nos ajuda a manter essa propriedade na inserção. Como em cada nível temos o primeiro valor estritamente menor que k, inserindo k logo depois desses valores, manterá a ordem de cada nível.

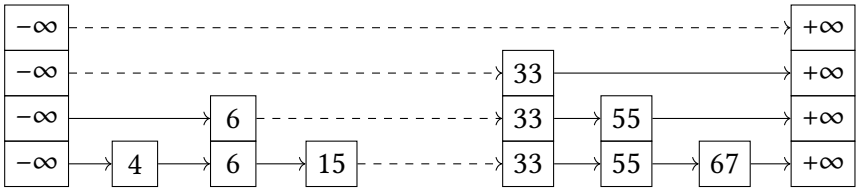


Figura 5.5: *Queremos inserir o valor 24 na estrutura. Assumindo que colocaremos esse valor em dois níveis, primeiro executamos `lesser(24)` para descobrir aonde inserir o valor*

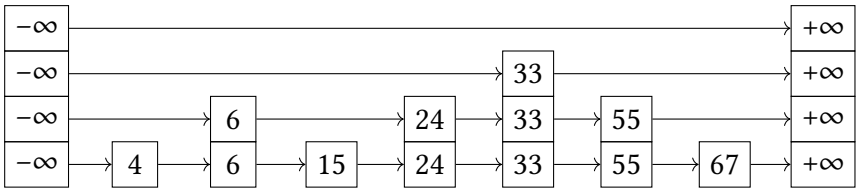


Figura 5.6: *Depois só precisamos adicionar esse valor nos primeiros dois níveis*

```
template <class T>
void SkipList<T>::insert(T key) {
    int level = __builtin_clz(rand());
    while (head->next.size() < level + 1) {
        head->next.push_back(NULL);
    }
    auto new_node = new SkipListNode<T>{key};
    new_node->next = std::vector<SkipListNode<T>*>(level);
    auto nodes = lesser(key);
    while (level--) {
        new_node->next[level] = nodes[level]->next[level];
        nodes[level]->next[level] = new_node;
    }
}
```

Para decidir em quantos níveis o novo elemento vai aparecer usamos `__builtin_clz(rand())`. `__builtin_clz` é uma função que conta quantos zeros a esquerda a representação binária de um `int` possui.

Note que essa implementação é limitada pelo tamanho da palavra `int` e pela qualidade da função `rand`. O importante aqui é que se o gerador de números aleatórios for bom o suficiente, cada bit do número se comporta independentemente, fazendo com que contar

zeros a esquerda seja equivalente a testar a presença do elemento em cada nível com probabilidade $p = \frac{1}{2}$.

Para qualquer probabilidade inversa de uma potência de 2, podemos utilizar uma sequência de bits aleatórios para gerar o nível de um elemento com facilidade.

Remoção

Antes de remover uma chave k, precisamos achar ele em cada um dos níveis. Assim como na busca, podemos fazer isso com `lesser`. Em cada nível, pulamos a chave k fazendo `lesser` naquele nível apontar diretamente depois de k.

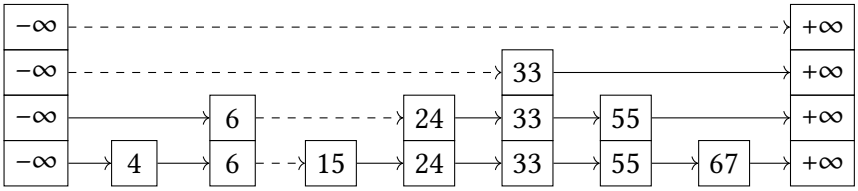


Figura 5.7: *Removendo o elemento 15 do exemplo anterior, olhando as arestas pontilhadas que saem dos valores de `lesser(15)`*

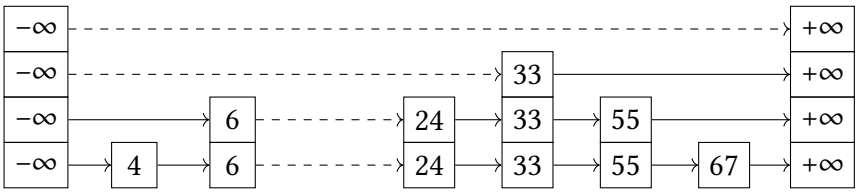


Figura 5.8: *Depois só precisamos pular o valor que precisa ser removido em cada nível*

```
template <class T>
void SkipList<T>::remove(T key) {
    auto nodes = lesser(key);
    int max_level = nodes.size();
    for (int level = 0; level < max_level; level++) {
        auto node = nodes[level]->next[level];
        if (node != NULL && node->key == key) {
            nodes[level]->next[level] = node->next[level];
        }
    }
}
```

5.3 Complexidade

A complexidade de espaço da skip list está relacionada a quantos nós aparecem no total, enquanto a complexidade de tempo está relacionada a o quão bem conseguimos espaçar os nós em cada nível. Essas duas propriedades foram escolhidas ao acaso durante a inserção dos elementos, e mesmo assim, a skip list é eficiente em ambos os critérios.

Espaço

Em uma skip list de n elementos, temos um primeiro nível de tamanho n. Para cada nível acima, o esperado é que apenas uma porcentagem p desses elementos estarão presentes, e isso continua para cada nível que subimos.

Podemos calcular a soma infinita da progressão geométrica com razão p para determinar quantos elementos teremos no total. Nossa quantidade esperada total de elementos é $\frac{n}{1-p}$, e logo, nossa complexidade de espaço é $O(n)$.

Tempo

Vamos calcular a complexidade da função `lesser`, que é utilizada por todas as nossas operações.

Para isso, realizaremos o processo inverso feito por `lesser(x)`, começando no primeiro nível, no maior valor que é estritamente menor que x e voltamos até a head. Sempre subimos um nível no mesmo elemento, ou pegamos uma aresta para a esquerda.

O final desse processo acontece quando estamos no último nível do primeiro elemento, que é o mesmo que subir tantas vezes quanto o nível máximo da skip list. Notamos assim que o valor esperado para a quantidade de operações feitas por `lesser` só depende da quantidade de níveis da skip list.

Também sabemos que precisamos subir sempre que existir um nível acima, dado que sempre andamos o máximo possível para a direta antes de descer um nível. Como a probabilidade de existir um nível acima do atual é p, temos probabilidade p de subir um nível, e probabilidade $1 - p$ de voltar no mesmo nível. Chamando $C(k)$ o custo para subir k níveis, temos que $C(0) = 0$ e

$$C(k) = (1 + p)C(k - 1) + (1 + (1 - p))C(k)$$

$$C(k) = \frac{1}{p} + C(k - 1)$$

$$C(k) = \frac{k}{p}$$

Como o valor esperado para o nível máximo em uma skip list de n elementos é $\log_{\frac{1}{p}} n$, temos que a complexidade de `lesser` é $O(\log n)$.

A complexidade da busca é igual a complexidade de `lesser`, e a inserção e remoção utilizam `lesser` e depois iteram por todos os níveis da skip list fazendo operações constantes, logo também possuem complexidade $O(\log n)$.

5.4 Acesso aleatório

As operações de busca, inserção e remoção em uma Skip List simples são todas $O(\log n)$ esperado, mas descobrir o k-ésimo elemento tem complexidade linear. Assim como nas

árvores balanceadas, uma pequena modificação nos permite encontrar o elemento em uma posição qualquer em complexidade logarítmica, assim como as outras operações da estrutura.

Além de guardar o próximo elemento naquele nível, cada elemento também vai guardar quantos *links* existem entre esses dois elementos no primeiro nível da estrutura.

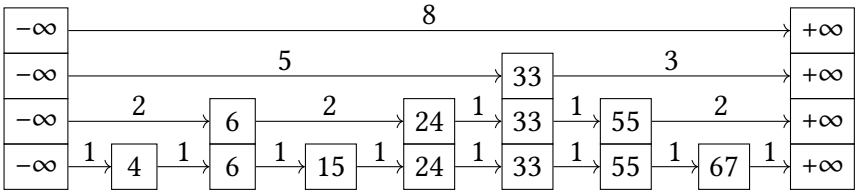


Figura 5.9: *Uma skip list que permite acesso aleatório em* $O(\log n)$

Para encontrar o k-ésimo menor elemento, fazemos a busca normal, dessa vez mantendo a soma dos links pulados, comparando com k. Nas as outras operações, precisamos atualizar os valores dos pulos na inserção e remoção, mas como os únicos valores que mudam são os das arestas retornadas por `lesser`, atualizar esses valores não muda a complexidade de nossas operações.

Agora que podemos encontrar o k-ésimo elemento, nossa skip list não precisa mais manter seus valores ordenados, contando que as operações de inserção e remoção trabalhem em cima de posições e não de números. Assim temos uma estrutura que pode representar uma lista, com inserção e remoção em uma posição e acesso aleatório, todos em $O(\log n)$.

5.5 Utilizações

A skip list tem uma implementação muito simples e consegue ser muito rápida. Além disso, ela mantém seus valores ordenados, algo que não é feito por outras estruturas simples como hash tables. Por esses benefícios, ela é muito utilizada em bancos de dados, como o Redis que usa skip lists em conjunto com hash tables (*Redis Source Code* 2021), e o Apache HBase (*Apache HBase Source Code* 2019).

Esse último utiliza uma versão concorrente da skip list implementada em Java (*ConcurrentSkipListMap Documentation* 2021). Existem outras estruturas com versões concorrentes, mas hash tables concorrentes não mantém os elementos ordenados, e versões concorrentes de árvores rubro-negras são complexas e só são mais eficientes que skip lists concorrentes em casos específicos (Besa e Eterovic, 2012).

Outro uso para skip lists acontece em sistemas distribuídos onde precisamos fazer consultas em range, impossibilitando o uso de estruturas baseadas em hash tables (Alam *et al.*, 2014).

Capítulo 6

Filtro de Bloom

Verificar se um elemento x pertence a um conjunto S é uma tarefa recorrente em qualquer sistema, O filtro de Bloom implementa essa tarefa com foco em utilizar memória eficientemente (Bloom, 1970).

Para fazer isso, a implementação básica do filtro de Bloom tem algumas restrições. Uma delas é que elementos não podem ser removidos, porque como veremos mais adiante, remover um elemento pode acabar com a integridade da estrutura. Outra restrição é que os elementos do conjunto S não podem ser recuperados, porque apenas uma representação simplificada deles é guardada na estrutura.

Com o representação simplificada dos elementos, existe uma chance de algum elemento y que não está em S ter a mesma representação de um elemento que está. Nesse caso, o filtro vai erroneamente afirmar que o elemento está em S, causando um falso positivo. Já falsos negativos não são possíveis, então quando o filtro responde que um elemento não está em S isso é garantidamente verdade. Podemos controlar essa probabilidade de falsos positivos no filtro de Bloom.

6.1 Estrutura

O filtro é representado por um vetor de bits de tamanho m. Inicialmente, todos os bits são 0.

```
struct BloomFilter {
    int m, n, k;
    double p;
    std::vector<bool> bits;
    BloomFilter(int, double);
    template <class T> void insert(T);
    template <class T> bool find(T);
    template <class T> size_t hash(T, size_t);
};
```

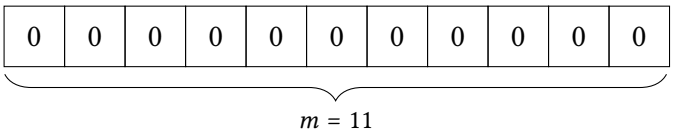


Figura 6.1: *Um filtro vazio de tamanho* 11

6.2 Operações

O filtro possui duas operações, `inserção` e `busca`. Nessas operações, `ligar` ou `desligar` um bit é, respectivamente, atribuir 1 ou 0 a esse bit.

Para realizar as operações também vamos precisar de k funções de hash $h_1, h_2, \ldots, h_k$. Essas funções recebem um elemento qualquer x e devolvem um valor no intervalo $[0, m-1]$, uma posição no vetor de bits. No momento vamos nos concentrar apenas em utilizar essas funções, mais adiante vamos mostrar como construir essas k funções uniformes e independentes.

Com isso, podemos construir nossas operações de inserção e busca.

Inserção

Para inserir um elemento x no filtro só precisamos calcular o valor das k funções de hash com x. Para cada função de hash h_i, vamos ligar a posição $h_i(x)$ do vetor de bits.

```
template <class T>
void BloomFilter::insert(T element) {
    for (int i = 0; i < k; i++)
        bits[hash(element, i)] = 1;
}
```

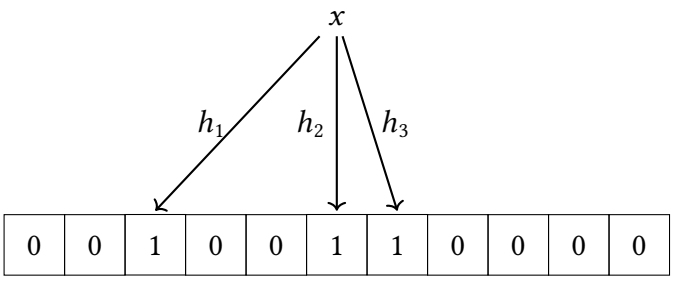


Figura 6.2: *Inserindo um elemento x utilizando $k = 3$ funções de hash*

Note que fazemos isso independente do estado nessa posição. Por exemplo, se vamos inserir um elemento y no filtro do exemplo anterior e $h_3(y) = h_2(x)$, essa posição no vetor continua ligada.

Busca

A busca é tão simples quanto a inserção: sabemos que quando um elemento x foi inserido ligamos todas as posições $h_i(x)$ do vetor de bits. Para verificar se x está no filtro

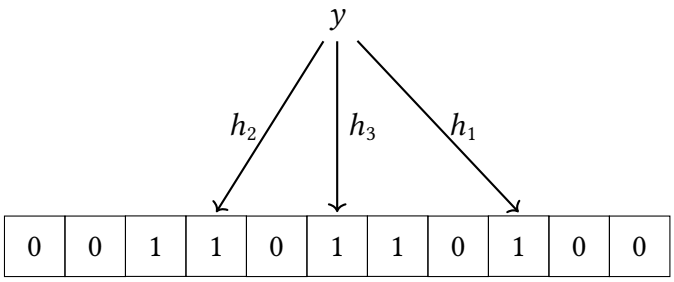


Figura 6.3: *Inserindo um elemento y após a inserção do elemento x*

vamos garantir que todas essas posições estão ligadas.

Se não for o caso e tivermos pelo menos uma posição desligada no vetor de bits, temos garantia de que esse valor nunca foi inserido.

Entretanto, se todas as posições estiverem ligadas pode ser que um conjunto de elementos que foram inseridos antes de x ligaram essas mesmas posições, o que cria a possibilidade de falsos positivos.

```
template <class T>
bool BloomFilter::find(T element) {
    bool found = true;
    for (int i = 0; i < k; i++)
        found &= bits[hash(element, i)];
    return found;
}
```

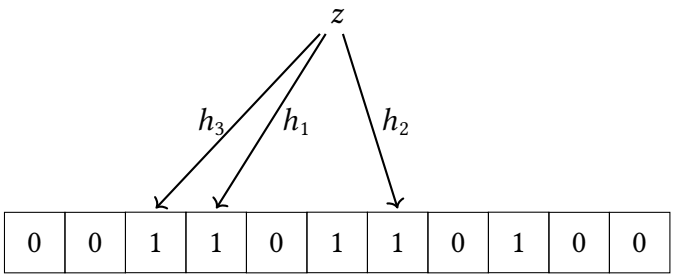


Figura 6.4: *Buscando um elemento z no filtro e recebendo um falso positivo. Note que nem x nem y são individualmente responsáveis por todos esses bits estarem ligados, mas sim os dois em conjunto.*

6.3 Encontrando valores ótimos

Para conseguirmos fazer observações e implementar essa estrutura, precisamos relacionar os valores de p, m, k e $n = |S|$. Vamos fazer isso a partir de uma aproximação para p.

Probabilidade de falsos positivos

Primeiro precisamos fazer um leve desvio para falar de uma aproximação para e^{-1}.

$$\lim_{m\to\infty}\left(1-\frac{1}{m}\right)^{m}=e^{-1}$$

Esse limite tende a e^{-1} rapidamente, de forma que até para valores baixos de m vamos conseguir aproximações bem decentes.

Voltamos ao calculo de p: considerando que as k funções de hash são independentes, temos que a probabilidade de uma posição do vetor de bits não ser escolhida na inserção de um elemento é:

$$\left(1-\frac{1}{m}\right)^{k}=\left(1-\frac{1}{m}\right)^{\frac{km}{m}}=\left(\left(1-\frac{1}{m}\right)^{m}\right)^{\frac{k}{m}}\approx e^{-\frac{k}{m}}$$

A probabilidade de uma posição estar **desligada** depois que todos os elementos de S foram inseridos é então

$$e^{-\frac{kn}{m}}$$

E a de estar **ligada** é

$$1-e^{-\frac{kn}{m}}$$

Para acontecer um falso positivo para a busca de um elemento x que não está em S, precisamos que todas as posições que serão verificadas pelas k funções de hash estejam ligadas, logo, a probabilidade de um falso positivo é:

$$p=\left(1-e^{-\frac{kn}{m}}\right)^{k}$$

Relacionando as variáveis

Dados n e m fixos, o valor de k que minimiza p é

$$k=\frac{m}{n}\ln 2$$

Agora conseguimos relacionar p e m da seguinte forma

$$p=\left(1-e^{-\left(\frac{m}{n}\ln 2\right)\frac{n}{m}}\right)^{\frac{m}{n}\ln 2}=\left(1-e^{-\ln 2}\right)^{\frac{m}{n}\ln 2}=\left(\frac{1}{2}\right)^{\frac{m}{n}\ln 2}$$
$$\Longrightarrow \ln p=-\frac{m}{n}(\ln 2)^{2}=-k\ln 2$$

Agora temos equações simples que relacionam todos as variáveis importantes para a construção do filtro de Bloom. Com dois desses valores conhecidos conseguimos descobrir

os outros.

```
BloomFilter::BloomFilter(int size, double err) {
    n = size;
    p = err;
    k = ceil(-log(p) / log(2));
    m = ceil((k * n) / log(2));
    bits = std::vector<bool>(m);
}
```

6.4 Construindo funções de hash

A criação de k funções de hash pode parecer algo custoso, mas na prática é possível conseguir resultados bem aceitáveis a partir de uma única função de hash h.

Se sua função de hash tem imagem grande o suficiente, é possível "quebrar"essa saída em k pedaços, cada um definindo uma das funções h_i.

Outra solução é concatenar algum valor ao elemento que será utilizado. Podemos, por exemplo, concatenar x e i antes de calcular $h(x)$, criando assim as funções $h_i(x) = h(x + i)$. Um cuidado que é preciso tomar com essa solução é garantir que não vamos acabar criando colisões triviais. Por exemplo, se tempos 11 funções de hash e queremos inserir a palavra A no filtro usando esse método de uma maneira simples, vamos tirar o hash das palavras A0, A1, A2, ..., A10. Note que a última palavra A10 também pode ser atingida pelo primeiro hash da palavra A1, criando uma colisão desnecessária. Uma forma de resolver isso é concatenar zeros até que o tamanho da palavra i seja igual para todos os hashes.

Outra solução muito simples é quebrar a saída do hash em apenas dois pedaços UPPER e LOWER e calcular h_i = UPPER + $i \times$ LOWER. Essa solução gera resultados satisfatórios e é usada na prática (*Guava BloomFilter* 2019).

```
template <class T>
size_t BloomFilter::hash(T element, size_t i) {
    std::hash<T> f;
    size_t result = f(element);
    size_t upper = (result >> HALF);
    size_t lower = result - (upper << HALF);
    return (upper + (i * lower)) % m;
}
```

`HALF` é definido como `4 * sizeof(size_t)` pois queremos quebrar o resultado da função de hash na metade. Note que implementações de C++ em que o tamanho de `size_t` não é grande podem afetar negativamente a performance do filtro.

6.5 Utilização de memória

Como foi mostrado na seção 6.3, podemos relacionar m/n ao erro:

$$\frac{m}{n} = -\frac{\ln p}{(\ln 2)^2} \approx -2.08 \ln p$$

Assim temos que a quantidade de bits muda linearmente com o logaritmo natural da probabilidade de falso positivo.

Para colocar isso em perspectiva, para utilizar 32 bits por elemento sua probabilidade de erro precisaria ser 2.1×10^{-7}

Outro jeito de observar como esse valor cresce devagar é notar que deixar essa probabilidade c vezes menor muda a quantidade de bits por elemento apenas por uma constante:

$$-2.08 \ln \frac{p}{c} = -2.08(\ln p - \ln c) = \frac{m}{n} + 2.08 \ln c$$

6.6 Adicionando mais elementos que o esperado

O valor de $n = |S|$ foi utilizado durante toda a seção 6.3 com a ideia de ser um valor fixo, mas o que acontece quando continuamos inserindo mais elementos do que tínhamos calculado na construção do filtro?

Inserindo c vezes mais elementos do que o esperado, temos

$$p' = e^{-\frac{m}{cn}(\ln 2)^2} = \left(e^{-\frac{m}{n}(\ln 2)^2}\right)^{\frac{1}{c}} = p^{\frac{1}{c}}$$

Para valores altos de p o valor não muda tão drasticamente, mas para valores perto de 0 (que são os mais comuns nos usos do filtro de Bloom), esse aumento pode causar problemas. Por exemplo, para $p = 0.05$, adicionar 10% a mais do que o esperado aumenta a probabilidade de falso positivo em 30%.

6.7 Remoção de elementos

O único jeito de remover um elemento do filtro descrito anteriormente é desligar pelo menos uma posição entre as k determinadas pelas funções de hash, o problema é que sem guardar informação adicional não é possível determinar se essa posição também foi ligada por outros elementos.

Se ela foi ligada por outro elemento, desligar essa posição criaria um falso negativo.

Um jeito de implementar remoção e amenizar os falsos negativos é construir um segundo filtro onde serão inseridos os elementos removidos do primeiro. Para verificar se um elemento está no conjunto precisamos que ele esteja no primeiro filtro e não no segundo. Note que essa solução introduz falsos negativos, dado que um falso positivo no filtro de remoção é um falso negativo para o problema original.

Outra maneira de habilitar a remoção de elementos é usar uma generalização do filtro de Bloom, chamada **counting bloom filter**.

Na inserção, no lugar de atribuir 1 as posições apontadas pelas funções de hash, incrementamos o valor na posição em 1. Note que agora, para que o counting bloom filter tenha qualquer utilidade, precisamos de mais que um bit para cada posição.

A busca continua parecida, só precisamos verificar se todas as posições são não nulas.

Com essas duas modificações simples, conseguimos implementar a remoção de um elemento. Basta fazer o contrário da função de inserção: decrementamos em 1 todas as posições apontadas pelas funções de hash.

O benefício do counting bloom filter é claro o suficiente: agora conseguimos remover elementos sem introduzir outro filtro ou falsos negativos. O malefício é que agora precisamos de múltiplos bits para cada posição, o que torna o uso de espaço bem mais custoso.

Para m posições, k funções de hash e n valores, a quantidade esperada de colisões entre as funções de hash é $\frac{kn}{m}$. Dependendo dos parâmetros ótimos para uma dada aplicação, usar uma quantidade de bits grande o suficiente para permitir esse valor com alguma folga por segurança pode não ser prático.

Podemos minimizar esse custo mantendo 1 bit por posição e adicionando mais bits apenas quando necessário, deixando o mesmo custo amortizado.

6.8 Paralelização

Como as k funções de hash são independentes, ambas as operações do filtro de bloom podem ser paralelizadas.

Além disso, podemos paralelizar operações do mesmo tipo, ou seja, fazer várias operações de inserir ao mesmo tempo, ou várias operações de busca ao mesmo tempo.

6.9 Utilizações

O filtro de Bloom é utilizado em diversas aplicações onde precisamos eficientemente descobrir se um elemento está em um conjunto.

O browser Chromium utilizava filtros de bloom para sua ferramenta de safe-browsing (*Chromium Bloom Filter Issue* 2011), mantendo um filtro de URLs maliciosos localmente. Nesse exemplo específico, não permitir que os usuários tenham acesso a uma lista de URLs maliciosos pode ser um benefício, nesse caso, a limitação do filtro de Bloom de não permitir a reconstrução do conjunto pode ser um ponto positivo.

Bancos de dados como PostgreSQL (*Postgres Source Code* 2021) e Apache Cassandra (*Cassandra Documentation* 2016) usam o filtro para minimizar as visitas ao disco rígido para entradas inexistentes.

O Amazon Redshift passou a utilizar bloom filter, notando consultas duas vezes mais eficientes. (*Amazon Redshift Announcement* 2020)

Capítulo 7

Comentários finais

É difícil captar o poder da aleatorização pois o seu uso costuma ser muito sutil. Nas estruturas apresentadas nesse texto, por exemplo, aleatorização existe em uma única linha do código, somente na inserção dos valores, e mesmo assim gera estruturas previsíveis com propriedades que ajudam nas outras operações.

Por outro lado, essa simplicidade é justamente o que torna aleatorização tão poderosa. Qualquer pessoa que já implementou uma árvore rubro-negra sabe que a tarefa de balancear uma árvore binária pode ser algo complicado, mas aleatorização atribui essa complexidade ao acaso, com resultados incríveis.

Os temas que foram abordados aqui são incrivelmente vastos, e só foi possível apresentá-los superficialmente. Com mais tempo, gostaria de falar mais sobre múltiplos assuntos.

Os detalhes do algoritmo do Mersenne Twister são muito complexos, e não tive tempo de estudá-los a fundo e comentá-los no texto.

Em testes de aleatorização, o teste espectral (Knuth, 1998) não foi abordado, mas é um ótimo teste genérico para aleatorização, que consegue cobrir múltiplos critérios, mas que também é bastante intricado.

Provas de maratona de programação utilizam aleatorização constantemente, de forma bem engenhosa, mas que não encaixavam bem na estrutura desse texto.

Em Geometria computacional, foram estudados diversos algoritmos e paradigmas de utilização de aleatorização abordados por Mulmuley (Mulmuley, 2002), dado que *aleatorização em geometria computacional* foi o primeiro título desse texto, que teve seu tema abrangido e parte desses algoritmos geométricos não foram tratados.

Espero, porém, que o que foi apresentado ajude a entender mais sobre aleatorização e sobre como podemos utilizá-la para resolver problemas em computação.

Referências

[*A million random digits: with 100.000 normal deviates* 1955] *A million random digits: with 100.000 normal deviates*. Free Press, 1955 (citado na pg. 3).

[Alam *et al.* 2014] Sarwar Alam, Humaira Kamal e Alan Wagner. "A scalable distributed skip list for range queries". Em: *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC 14* (2014). doi: 10.1145/2600212.2600712 (citado na pg. 27).

[*Amazon Redshift Announcement* 2020] *Amazon Redshift Announcement*. 2020. url: https://aws.amazon.com/about-aws/whats-new/2020/05/amazon-redshift-now-leverages-bloom-filters-to-improve-data-lake-query-performance/ (citado na pg. 36).

[*Apache HBase Source Code* 2019] *Apache HBase Source Code*. 2019. url: https://github.com/postgres/postgres/blob/ca3b37487be333a1d241dab1bbdd17a211a88f43/src/backend/lib/bloomfilter.c (citado na pg. 27).

[Besa e Eterovic 2012] Juan Besa e Yadran Eterovic. "A concurrent red black tree". Em: *Parallel and Distributed Computing and Systems* (2012). doi: 10.2316/p.2012.757-056 (citado na pg. 27).

[Bloom 1970] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". Em: *Communications of the ACM* 13.7 (1970), pgs. 422–426. doi: 10.1145/362686.362692 (citado na pg. 29).

[Carlisle 2009] Rodney Carlisle. *Encyclopedia of play in today's society*. SAGE Publications, 2009, pgs. 171–173 (citado na pg. 1).

[*Cassandra Documentation* 2016] *Cassandra Documentation*. 2016. url: https://github.com/apache/cassandra/blob/79e693e16e2152097c5b27d2d7aaa1763e34f594/doc/source/operating/bloom_filters.rst (citado na pg. 35).

[*Chromium Bloom Filter Issue* 2011] *Chromium Bloom Filter Issue*. 2011. url: https://bugs.chromium.org/p/chromium/issues/detail?id=71832 (citado na pg. 35).

[Clarkson 1987] K. L. Clarkson. "New applications of random sampling in computational geometry". Em: *Discrete & Computational Geometry* 2.2 (1987), pgs. 195–222. doi: 10.1007/bf02187879 (citado na pg. 11).

[Clarkson 1988] K. L. Clarkson. "Applications of random sampling in computational geometry, ii". Em: *Proceedings of the fourth annual symposium on Computational geometry - SCG 88* (1988). doi: 10.1145/73393.73394 (citado na pg. 11).

[*ConcurrentSkipListMap Documentation* 2021] *ConcurrentSkipListMap Documentation*. 2021. url: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html (citado na pg. 27).

[*Guava BloomFilter* 2019] *Guava BloomFilter*. 2019. url: https://github.com/google/guava/blob/9ef8b741ce47b1b5836e4c2d4b4004ee599f5cef/guava/src/com/google/common/hash/BloomFilterStrategies.java (citado na pg. 33).

[Haahr s.d.] Mads Haahr. *True Random Number Service*. url: https://www.random.org/ (citado na pg. 3).

[Knuth 1998] D. E. Knuth. *The art of computer programming Volume 2: seminumerical algorithms*. Addison-Wesley, 1998, pgs. 93–118 (citado na pg. 37).

[L'ecuyer e Simard 2007] Pierre L'ecuyer e Richard Simard. "Testu01". Em: *ACM Transactions on Mathematical Software* 33.4 (2007), pgs. 1–40. doi: 10.1145/1268776.1268777 (citado na pg. 6).

[Marsaglia 1995] George Marsaglia. *The Marsaglia random number CDROM: including the diehard battery of tests of randomness*. 1995. url: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/ (citado na pg. 6).

[Matsumoto e Nishimura 1998] Makoto Matsumoto e Takuji Nishimura. "Mersenne twister". Em: *ACM Transactions on Modeling and Computer Simulation* 8.1 (1998), pgs. 3–30. doi: 10.1145/272991.272995 (citado na pg. 5).

[Mulmuley 2002] Ketan Mulmuley. *Computational geometry: an introduction through randomized algorithms*. Prentice Hall, 2002 (citado na pg. 37).

[*Postgres Source Code* 2021] *Postgres Source Code*. 2021. url: https://github.com/postgres/postgres/blob/ca3b37487be333a1d241dab1bbdd17a211a88f43/src/backend/lib/bloomfilter.c (citado na pg. 35).

[Pugh 1990] William Pugh. "Skip lists: a probabilistic alternative to balanced trees". Em: *Communications of the ACM* 33.6 (1990), pgs. 668–676. doi: 10.1145/78973.78977 (citado na pg. 21).

[*Redis Source Code* 2021] *Redis Source Code*. 2021. url: https://github.com/redis/redis/blob/62b1f32062b8f688179a8262959a5b80d0ad4de7/src/t_zset.c#L40 (citado na pg. 27).

[Tippett e Pearson 1959] L. H. C. Tippett e Karl Pearson. *Random sampling numbers*. Cambridge University, 1959 (citado na pg. 3).