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

Uma Coletânea de Programming Pearls

Victor Chiaradia Gramuglia Araujo

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisor: Prof. Dr. Carlos Eduardo Ferreira

São Paulo
24 de Dezembro de 2021

Resumo

Victor Chiaradia Gramuglia Araujo. **Uma Coletânea de Programming Pearls**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Existem incontáveis trechos de código considerados geniais, dentre os mais importantes, os algoritmos e estruturas de dados fundamentais são estudados por alunos de graduação do mundo inteiro. Entretanto, muitos outros deixam de ser abordados, pois existe uma quantidade finita de tempo na graduação de um aluno. Um graduando é capaz dele mesmo destrinchar artigos e código-fonte, analisando a performance dos algoritmos em diversos casos e decidir se sua utilização seria benéfica em um determinado contexto prático. Dito isso, neste trabalho serão apresentados ao leitor três algoritmos, o *Fast Inverse Square Root*, o *Completely Fair Scheduler* e o *Hierarchical Path-Finding A**, além de quatro estruturas de dados, o *Bloom Filter*, a *XOR Linked List*, a *B+ tree* e a *Piece Tree*.

Palavras-chave: Programming Pearls. Algoritmos. Estruturas de Dados. Coletânea.

Abstract

Victor Chiaradia Gramuglia Araujo. **A Collection of Programming Pearls**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

There are uncountable pieces of code that are considered genius, among the most important, we have the algorithms and data structures that are taught in undergrad computer science courses around the world. However, many more are not covered, since there is only a finite amount of time during a student's graduation. An undergrad student has the capability to unravel papers and source codes, to make a deep analysis of an algorithm's performance and judge if that algorithm would be beneficial in a specific context. With that in mind, this work will present to the reader three algorithms, the *Fast Inverse Square Root*, the *Completely Fair Scheduler* and the *Hierarchical Path-Finding A**, and also four data structures, the *Bloom Filter*, the *XOR Linked List*, the *B+ tree* and last but not least, the *Piece Tree*.

Keywords: Programming Pearls. Algorithms. Data Structures. Collection.

Sumário

Capítulo 1

Introdução

Alunos de graduação aprendem diversas estruturas de dados e algoritmos durante seu tempo na universidade, por exemplo: *Bubble Sort*, *Quicksort*, filas, pilhas, árvores de busca binária, árvores rubro-negras, busca em largura, algoritmo de Dijkstra e muitos outros. Entretanto, não é exagero dizer que incontáveis algoritmos e estruturas deixam de ser explorados, seja por uma questão de tempo, complexidade, obscuridade ou por não serem tão fundamentais para se construir uma boa base quanto os que são abordados.

Tendo isso em vista, este trabalho tem como objetivo a exploração de alguns elementos desse conjunto vasto de algoritmos e estruturas que não foram tratados durante a graduação.

A estrutura dos capítulos segue um modelo simples: primeiro, é feito uma introdução ao problema que a estrutura ou algoritmo deseja resolver. São apresentadas noções consideradas importantes para o melhor entendimento do assunto, sejam técnicas usadas anteriormente para atacar o problema ou definições de termos. Com o conhecimento necessário introduzido ao leitor, finalmente é feita uma explicação de como o algoritmo/estrutura de dados em foco funciona. Por último, é feito um resumo do que foi abordado no capítulo.

A escolha dos assuntos tratados não segue um padrão forte, porém, em geral, compartilham as seguintes características: não foi abordado em alguma disciplina obrigatória do Bacharelado em Ciência da Computação (BCC) do Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP); está relacionado com algum tópico explorado nas diversas disciplinas do BCC. Finalmente, foi dado uma maior ênfase para aqueles algoritmos/estruturas encontrados em *software* real.

Capítulo 2

Fast Inverse Square Root

2.1 Introdução ao problema

Em muitos momentos, na computação não é necessário achar uma solução exata de um problema, mas somente uma aproximação que possui um erro considerado admissível para a aplicação em questão. Em videogames, onde milhares de cálculos devem ser computados a cada segundo, a precisão dos cálculos deve ser balanceada com a sua rapidez. Por exemplo, para simular raios de luz em computação gráfica 3D é comum usar vetores normalizados,

$$\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}$$

no caso em que $\vec{v} \in \mathbb{R}^3$ pode ser escrito como,

$$\hat{v} = \frac{\vec{v}}{\sqrt{v_i^2 + v_j^2 + v_k^2}}$$

para calcular o ângulo que os raios de luz refletem e incidem em superfícies.

Por muitos anos as operações de divisão e o cálculo da raiz quadrada eram operações consideradas lentas. Assim, um passo custoso para a normalização de um vetor é fazer o cálculo de $\frac{1}{\sqrt{x}}$, onde x é a soma dos quadrados dos componentes do vetor. Então, surgiu um algoritmo que calcula uma aproximação para o inverso da raiz quadrada, tal algoritmo é comumente chamado de Fast Inverse Square Root (Fast InvSqrt).

Embora ainda não seja muito claro quem foi o primeiro a criar esse algoritmo, Sommefeldt, 2006a, traçou a origem do algoritmo até Greg Walsh em 1987. Devido à popularidade do jogo *Quake 3: Arena*, quando esse teve seu código-fonte publicado em 2005 sob a GNU GENERAL PUBLIC LICENSE Version 2 (GPLv2), o algoritmo acabou ganhando popularidade entre desenvolvedores de jogos e entusiastas. É justamente a implementação do Fast InvSqrt encontrada em *Quake 3: Arena* que iremos utilizar para nossa análise.

2.2 IEEE 754-1985

Para começar a entender o Fast InvSqrt, primeiro é necessário se familiarizar com os números float da linguagem C, mais especificamente em como eles são armazenados em memória. Um float em C é um número de ponto flutuante de precisão única, e sua representação em memória é definida pelo padrão IEEE 754. Embora não existam grandes diferenças nas versões do IEEE 754, esta análise usará a versão do padrão de 1985, essa sendo a versão vigente quando o jogo em questão foi lançado.

Um número de ponto flutuante é armazenado em memória usando notação científica, $f = s * M * 2^E$, onde s representa o sinal de f, M a mantissa e E o expoente. Um número flutuante de precisão única requer 32 bits de memória para armazenar tais informações. Essas informações são representadas da seguinte forma:

s	$E_7E_6...E_1E_0$	$M_{22}M_{21}...M_1M_0$

Tabela 2.1: *Ilustração de um float em memória*

onde para representar s se utiliza somente 1 bit s_0, com o valor 1 para números negativos e 0 para números positivos. Os próximos 8 bits são usados para representar E, em vez de usar o comum complemento de dois para representar números negativos (os bits de um número são invertidos e é somado 1 bit), é usado uma técnica chamada de offset binário. Nos 8 bits são armazenados $E - b$, sendo que b é chamado de *bias* e nesse caso possui valor igual a 127. Por fim, os próximos 23 bits armazenam a mantissa M. Vale ressaltar que números na base 2, representados em notação científica, sempre têm seu dígito mais significativo com o valor de 1, assim tal informação não precisa ser armazenada e pode ser inferida.

Seja f um número float qualquer, podemos o representar por

$$f = (-1)^s (1 + m) 2^{E-127} \tag{1.2.1}$$

onde m representa $\frac{M}{2^{23}}$.

2.3 Método de Newton–Raphson

O método de Newton–Raphson, também chamado de método de Newton, é um método numérico para encontrar raízes aproximadas de equações. Para encontrar os valores de x tal que $f(x) = 0$, primeiro é necessário achar uma aproximação inicial x_1, que pode ser encontrada usando qualquer método disponível. Podemos utilizar a seguinte fórmula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Supondo que $f'(x_i) \neq 0$, podemos usar esta fórmula para achar uma aproximação para uma raiz de $f(x)$. É importante mencionar que não é garantido que $\lim_{n \to +\infty} x_n$ irá convergir para uma raiz para qualquer x_1 inicialmente escolhido. Se for o caso da aproximação perder precisão com o passar das iterações, é preciso parar o processo e fazer uma nova escolha de x_1.

2.4 O algoritmo

```
1   float Q_rsqrt( float number )
2   {
3       long i;
4       float x2, y;
5       const float threehalfs = 1.5F;
6
7       x2 = number * 0.5F;
8       y = number;
9       i = * ( long * ) &y;
10      i = 0x5f3759df - ( i >> 1 );
11      y = * ( float * ) &i;
12      y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
13  // y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can be
        removed
14
15      return y;
16  }
```

Programa 2.1: *Implementação da função em Quake 3: Arena*

O Fast InvSqrt (chamado de Q_rsqrt no *Quake 3: Arena*) é uma aplicação do método de Newton-Raphson. Então, iremos começar nossa análise sobre como o x_1 é escolhido.

Em C, é possível manipular diretamente os bits de números inteiros (int, long, short etc), porém tais manipulações não são permitidas para números de ponto flutuante. Para contornar essa limitação, na linha 9 do código são feitos dois *casts*. Os bits de y são interpretados como os bits de um número inteiro e é armazenado o resultado dessa conversão em i. É importante mencionar que em C, tanto o tipo float como o tipo long possuem 32 bits.

Com a conversão de float para long realizada, um deslocamento de bits para a direita é feito em i na linha 10. Lembrando do padrão IEEE 754, o resultado desse deslocamento de bits é:

	s	E	M
i	s_0	$E_7E_6E_1E_0$	$M_{22}M_{21}M_1M_0$
i >> 1	0	$s_0E_7...E_2E_1$	$E_0M_{22}...M_2M_1$

Tabela 2.2: *Ilustração de um deslocamento de bits em um float*

Como desejamos tirar a raiz quadrada do argumento da função, podemos o considerar como positivo, já que estamos lidando com distâncias, assim o bit s_0 é sempre 0, então fazer o deslocamento de bits não irá impactar o sinal. No caso do expoente E, o deslocamento de

bits para a esquerda em um inteiro resulta na divisão inteira por dois. Dividir o expoente de um número por dois é equivalente a tirar a raiz quadrada desse número, porém como é armazenado $E - 127$ em memória, também dividimos o viés por dois. Já no caso da mantissa, ao fazer o deslocamento de bits perdemos o seu bit menos significativo M_0 e adicionamos E_0 como bit mais significativo. Se E_0 for 0 estamos simplesmente fazendo a divisão inteira por 2 em M, caso contrário estamos introduzindo um erro de no máximo 2^{22} ao valor da mantissa. Após o deslocamento de bits é usada a constante 0x5f3759df. Tal constante tem como papel corrigir o viés do expoente e minimizar o erro da mantissa para gerar um melhor palpite inicial para o método de Newton. Embora não se tenha certeza como os autores do algoritmo chegaram nessa constante, Robertson, 2012 analisa os efeitos que a constante tem sobre o método de Newton e introduz uma nova constante 0x5f375a86 que gera o menor erro possível.

Usamos o mesmo método de conversão de float para long em ordem reversa, para trabalharmos novamente com float. Assim temos armazenado na variável y o primeiro candidato x_1 ao resultado de $\frac{1}{\sqrt{x}}$, agora basta aplicar o passo iterativo do método de Newton-Raphson até encontrarmos uma aproximação suficiente próxima.

Para achar o resultado de $\frac{1}{\sqrt{x}}$, usaremos a equação,

$$f(y) = \frac{1}{y^2} - x$$

que possui derivada,

$$f'(y) = -\frac{2}{y^3}$$

então, cada passo do método de Newton terá forma,

$$y_{n+1} = y_n - \frac{\frac{1}{y_n^2} - x}{-\frac{2}{y_n^3}}$$

que pode ser reescrito como,

$$y_{n+1} = \frac{1}{2} y_n (3 - y_n^2 x)$$

ou, para estar mais próximo de como está representado no código,

$$y_{n+1} = y_n (1.5 - (\frac{x}{2} y_n y_n))$$

Embora o Fast InvSqrt geralmente utiliza dois passos do método de Newton, como pode ser observado pelo comentário na linha 13, os desenvolvedores de Quake 3 optaram por usar somente uma iteração. Pelo comentário, podemos especular que o erro com somente uma iteração era pequeno o suficiente para a aplicação deles.

2.5 Análise moderna de performance

Em Robertson, 2012, é apresentado a razão de tempo entre o método mais ingênuo $y = 1/sqrt(x)$ (rsqrt) e o Fast InvSqrt. Nas CPUs utilizadas nestes testes foi observado que o Fast InvSqrt era de de 3 até 4 vezes mais rápido do que o método ingênuo. É natural se perguntar se o Fast InvSqrt ainda é uma opção mais rápida do que o rsqrt. Com essa dúvida em mente, foram realizados testes com todos os números floats de 0 até o INFINITY de C e também todos os números floats de 1 até 100000.

Intel® Core™ i5-7400 e GCC 11.1.0		
Algoritmo	0.0f até INFINITY	1.0f até 100000.0f
Q_rsqrt	46.0299 segundos	2.94265 segundos
rsqrt	15.4862 segundos	0.9452 segundos

Tabela 2.3: *Comparação de tempo em segundos entre os algoritmos*

Intel® Xeon® CPU E5-2670 e GCC 8.3.0		
Algoritmo	0.0f até INFINITY	1.0f até 100000.0f
Q_rsqrt	64.2442 segundos	3.4837 segundos
rsqrt	24.2672 segundos	1.43426 segundos

Tabela 2.4: *Comparação de tempo em segundos entre os algoritmos*

Podemos concluir com os dados acima que, o Fast InvSqrt em sua forma original não permanece sendo uma opção mais rápida em CPUs modernas comuns e deve ser evitado. Isso se deve ao fato de que com a introdução das instruções DIVSS e SQRTSS, CPUs se tornaram mais eficientes em fazer ambos os cálculos de divisão e raiz quadrada com números de ponto flutuante de precisão única respectivamente.

No entanto, esses resultados não tornam o Fast InvSqrt uma relíquia do passado. Modificações no algoritmo ainda são publicadas nos dias atuais, isso é devido ao fato de que ainda existem microcontroladores que não possuem as instruções que deixam, em CPUs modernas convencionais, o Fast InvSqrt obsoleto.

2.6 Conclusão

Neste capítulo observamos como um conhecimento aprofundado do padrão IEEE 754, de cálculo e de certas funcionalidades da linguagem C. Tais técnicas permitiram que por muitos anos, desenvolvedores pudessem computar o inverso da raiz quadrada de uma maneira mais rápida do que usando métodos mais convencionais. A introdução de instruções específicas para a divisão e o cálculo da raiz quadrada de número de ponto flutuante diminui drasticamente o número de casos onde o Fast Inverse Square Root é superior à implementação mais simples. Mesmo com a queda de performance em CPUs modernas o algoritmo ainda continua sendo um exemplo histórico que ilustra como conhecimentos de áreas diversas podem ser usados para extrair um nível maior de performance.

Capítulo 3

XOR Linked List

3.1 Introdução ao problema

Nos anos em que computadores possuíam poucos kilobytes de memória RAM, era comum desenvolvedores usarem truques engenhosos para armazenar o máximo de informação em tão pouco espaço.

Neste capítulo iremos explorar uma maneira em que o operador XOR (ou exclusivo) pode ser usado para diminuir o uso de memória em uma lista duplamente ligada. Tal estrutura é chamada de XOR linked list e usa o operador ou exclusivo para guardar dois ponteiros em uma só variável.

3.2 Ou exclusivo

O operador *ou exclusivo*, também conhecido como *xor*, é um operador binário representado por ^ em C e muitas outras linguagens de programação e representado por $\oplus$ na matemática. Para cada bit com mesmo valor nas duas entradas, o bit resultante será igual a 0, caso os bits sejam diferentes o bit resultante será 1. Segue uma tabela para ilustrar tal comportamento.

	1	**0**
1	0	1
0	1	0

Tabela 3.1: *Tabela verdade do xor*

O xor possui algumas propriedades essenciais para entender o XOR linked list. O operador é comutativo, associativo, possui 0 como identidade e um elemento é seu próprio inverso.

$$A \oplus B = B \oplus A \tag{2.2.1}$$

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C \tag{2.2.2}$$

$$A \oplus 0 = A \tag{2.2.3}$$

$$A \oplus A = 0 \tag{2.2.4}$$

3.3 A estrutura de dados

A XOR linked list é um tipo de lista duplamente ligada, ou seja, cada nó da lista possui uma referência para seu antecessor e para seu sucessor. A grande diferença entre uma XOR linked list e uma lista duplamente ligada convencional é que em uma XOR linked list, ambas as referências dos vizinhos de um nó são armazenadas na mesma variável.

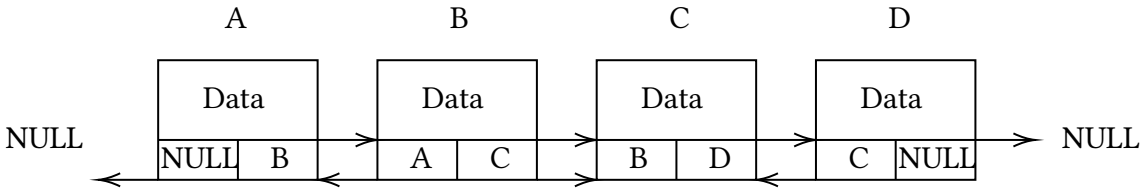

Figura 3.1: *Ilustração de um Lista duplamente ligada com 4 nós*

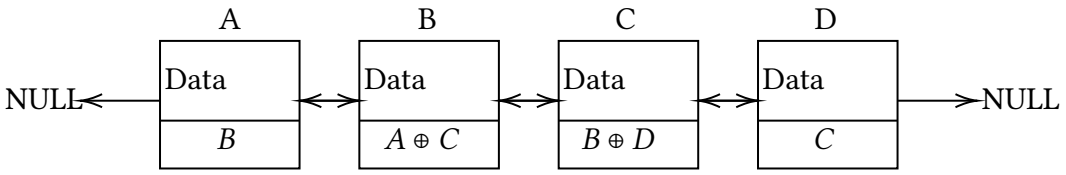

Figura 3.2: *Ilustração de um XOR linked list com 4 nós*

Observando o diagrama acima, a cabeça da lista (a primeira célula) aparenta possuir somente uma referência para seu sucessor. Porém, usando a equação (2.2.3) podemos interpretar que a cabeça também guarda uma referência a seu antecessor, pois o valor 0 nessa ocasião pode ser interpretada como uma referência para NULL (o antecessor da cabeça). O mesmo pode ser feito para a cauda da lista (sua última célula). Assim temos o invariante de que, toda célula possui armazenado em uma variável o xor dos endereços de seu antecessor e de seu sucessor.

Para estudar essa estrutura de dados, iremos analisar a implementação em pseudocódigo de três funções: a função de inserção no começo da lista, a função de remoção do primeiro

nó que possui um certo valor e uma função que insere um novo nó após um nó específico. Por questão de brevidade, iremos nos limitar a essas três funções pois acreditamos que elas encapsulam bem o comportamento da estrutura.

```
struct node { data, xor_pointer }

Function insert (head, data)
   if (head = null) then
      head ⟵ node(data, null)
   else
      new_node ⟵ node(data, head)
      head->xor_pointer ⟵ head->xor_pointer xor new_node
      head ⟵ new_node
   endif
   return
```

```
Function delete (head, data)
   prev ⟵ null
   current ⟵ head
   next ⟵ prev xor current->xor_pointer
   while (current ≠ null)
      if (current->data = data)
         if (prev ≠ null)
            prev->xor_pointer ⟵ prev->xor_pointer xor current xor next
         endif
         if (next ≠ null)
            next->xor_pointer ⟵ next->xor_pointer xor current xor prev
         endif
         if (current = head)
            head ⟵ next
         endif
         return
      endif
      prev ⟵ current
      current ⟵ next
      next ⟵ prev xor current->xor_pointer
   return
```

```
Function insert after(head, x, data)
   new_node ⟵ node(data, null)
   prev ⟵ null
   current ⟵ head
   next ⟵ prev xor current->xor_pointer
   while (current ≠ null)
      if (current = x)
         next->xor_pointer ⟵ next->xor_pointer xor current xor new_node
         current->xor_pointer ⟵ current->xor_pointer xor next xor new_node
         new_node->xor_pointer ⟵ = current xor next
         return
      endif
      prev ⟵ current
      current ⟵ next
      next ⟵ prev xor current->xor_pointer
   return
```

Começando pela função que cria um novo nó que armazena *data* e o insere no começo da

lista, chamada de *insert*. A parte que mais difere da implementação de uma lista duplamente ligada comum é a atualização do ponteiro dos vizinhos da cabeça da lista (*xor_pointer*). Para fazer tal atualização, se usa o operador xor, caso a cabeça tenha um vizinho para a esquerda, após a operação, será armazenado o xor do novo nó com o nó direito da cabeça em *xor_pointer*. No caso de a lista só possuir a cabeça, *xor_pointer* será null, já que a cabeça não possui vizinhos, assim pela equação (2.2.3) o endereço do novo nó será armazenado em *xor_pointer*.

A função *delete* irá remover da lista o primeiro nó que armazena *data*. Para cada iteração do laço é necessário guardar uma referência para o nó anterior *prev* (que se inicia como null), do nó atual *current* e do próximo nó *next*. Caso *current* armazene *data* iremos removê-lo da lista, para isso é necessário atualizar *xor_pointer* de *prev* e *next*, para que eles se conectem. Ao aplicar xor em *xor_pointer* de *prev* com *current*, iremos remover a referência de *current*,

$$(A \oplus B) \oplus A \overset{(2.2.1)}{\iff} (B \oplus A) \oplus A \overset{(2.2.2)}{\iff} B \oplus (A \oplus A) \overset{(2.2.4)}{\iff} B \oplus (0) \overset{(2.2.3)}{\iff} B$$

então basta fazer um xor do resultado com *next* para ligar *prev* e *next*. Vale ressaltar que devido a (2.2.2), não importa a ordem que essas operações sejam realizadas. Com *prev* e *next* devidamente conectados, *current* foi removido da lista. É necessário checar se é preciso atualizar a cabeça da lista caso a tenhamos removido.

Por fim, a função *insert_after* irá criar um novo nó e irá inseri-lo após *x*. Ao contrário de uma lista ligada convencional, não basta apenas possuir uma referência a *x* e fazer uma rápida mudança de variáveis. Precisamos também de pelo menos uma referência a um de seus vizinhos para poder fazer as operações de xor, de maneira similar na função de deleção. Por esse motivo é necessário percorrer toda a lista até encontrar *x*, mantendo uma referência ao nó anterior, com isso teremos uma referência ao antecessor de *x* e será possível inserir um nó após *x*.

3.4 Simulação

Para ajudar na compreensão da estrutura foram feitas algumas ilustrações de simulações de chamadas das funções *delete* e *insert after*.

Primeiro iremos simular a função *delete*, com a deleção de um elemento que possui armazenado um valor igual a 10. No começo da iteração do laço temos que: $prev = NULL$, $current = A$ e $nxt = B$.

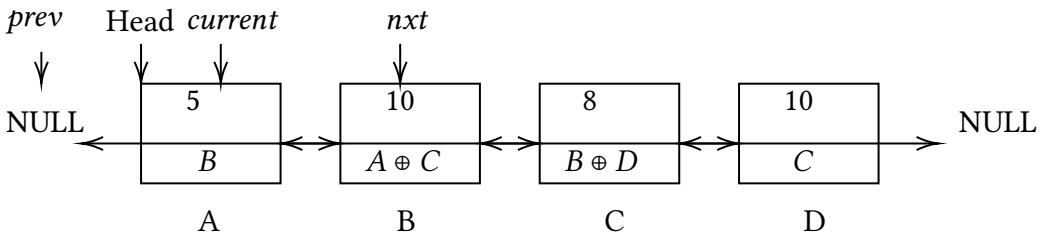

Figura 3.3: *Ilustração de uma chamada de delete(A, 10)*

Ao final da iteração teremos que, *prev* = *A*, *current* = *B* e por fim *nxt* = $A \oplus (A \oplus C) = C$. Assim a próxima iteração irá começar da seguinte forma:

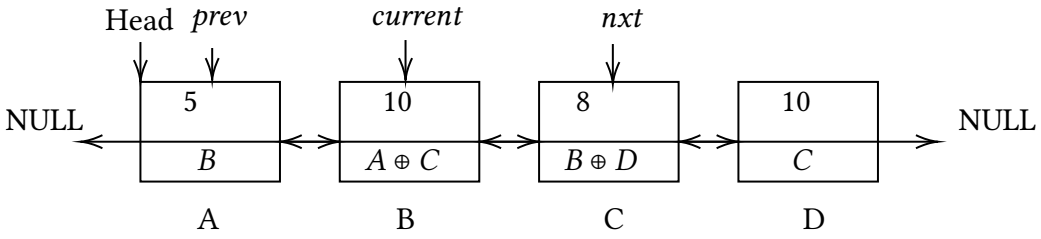

Figura 3.4: *Ilustração de uma chamada de delete(A, 10)*

Como o valor de *current* é igual ao valor que desejamos deletar, os ponteiros para os vizinhos de *prev* e *nxt* devem ser alterados. As seguintes alterações serão feitas: *prev->xor_pointer* = $B \oplus B \oplus C = C$ e *nxt->xor_pointer* = $(B \oplus D) \oplus B \oplus A = A \oplus D$. Assim a lista se encontra na seguinte forma no final da execução:

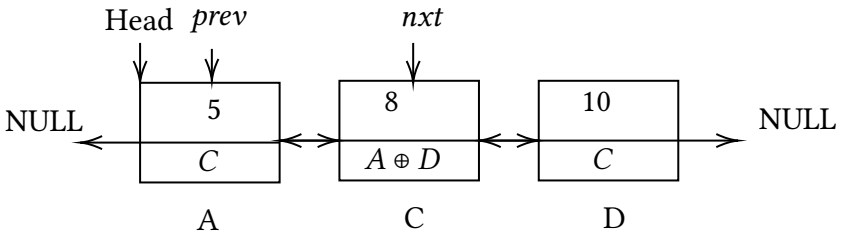

Figura 3.5: *Ilustração de uma chamada de delete(A, 10)*

A próxima simulação será da função *insert after*, onde iremos criar um novo nó que irá conter um valor igual a 20 e será inserido após o nó *B*. No começo da primeira iteração do laço, a função se encontrará no seguinte estado:

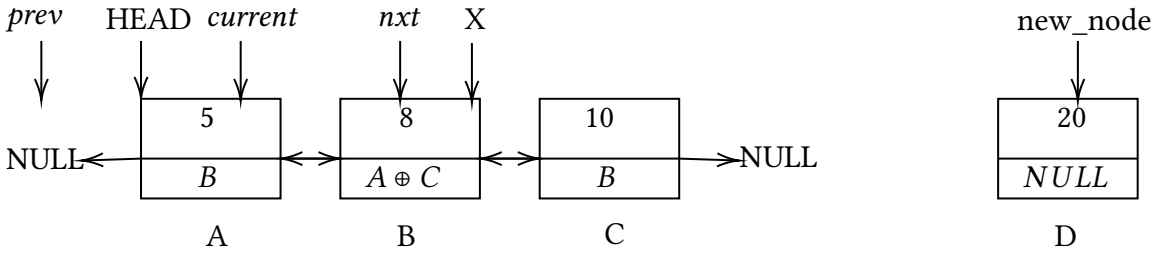

Figura 3.6: *Ilustração de uma chamada de insert after(A, B, 20)*

No final da primeira iteração, temos que *prev* = *A*, *current* = *B* e *nxt* = $(A \oplus C) \oplus A = C$. Assim, no começo da próxima iteração do laço, o algoritmo se encontrará no seguinte estado:

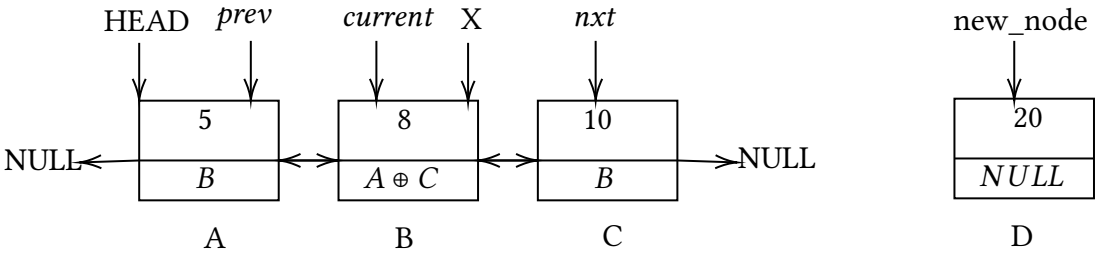

Figura 3.7: *Ilustração de uma chamada de insert after(A, B, 20)*

Durante esta iteração, *current* é igual a *X*, assim iremos inserir o novo nó (*new_node*) após *current*. Iremos atualizar os ponteiros para os vizinhos de *current*, *nxt* e *new_node* da seguinte forma: $current->xor\_pointer = (A \oplus C) \oplus C \oplus D = A \oplus D$, $nxt->xor\_pointer = B \oplus (B \oplus D) = D$ e $new\_node->xor\_pointer = B \oplus C$. No fim da chamada da função, a lista se encontra da seguinte forma:

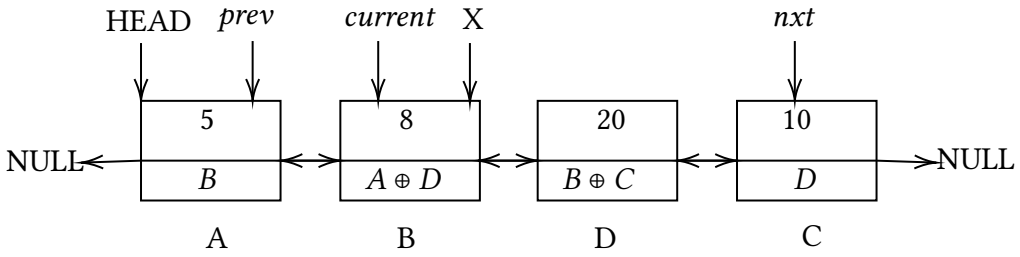

Figura 3.8: *Ilustração de uma chamada de insert after(A, B, 20)*

3.5 Análise de performance

Para medir a performance da estrutura, foi implementada em C, versões simples de uma lista duplamente ligada (dll) e uma XOR linked list (xll). Para coletar os dados seguintes foi usado uma CPU Intel® Core™ i5-7400, GCC 11.1.0 e Valgrind 3.17.0.

Começando pelo impacto na memória, é possível fazer um simples cálculo para a diferença de uso de memória entre as duas estruturas.

tamanho de um ponteiro * número de nós

Embora o tamanho de ponteiros seja dependente da máquina que está sendo utilizada, máquinas modernas geralmente utilizam 8 bytes para armazenar um ponteiro. Para verificar a fórmula, foram criadas duas listas com 10000000 nós e foi usado o programa Valgrind para fornecer o total de memória usada. A lista duplamente ligada tradicional teve um total de 240000000 bytes (240 MB) alocados, enquanto a XOR linked list teve um total de 160000000 bytes (160 MB) alocados. A diferença de memória usada foi de 80 MB, o que está de acordo com a fórmula usando um tamanho de ponteiro igual a 8 bytes e 10000000 itens, uma diferença de um terço. É preciso mencionar que a diferença relativa entre as duas implementações depende de quantos bytes são usados para armazenar os dados de cada nó, quanto mais memória for usada para armazenar os dados, menor será a diferença relativa.

Para medir o impacto no tempo de execução foram realizados os seguintes testes: a inserção de 100000 nós na cabeça da lista, a deleção de 10000 nós aleatórios em uma lista de 100000 nós e a inserção de 10000 nós em posições aleatórias de uma lista com originalmente 100000 nós.

	xll	dll
insert	0.016 s	0.015 s
delete	2.96 s	2.19 s
insert_after	3.03 s	0.000529 s

Tabela 3.2: *Comparação de tempo em segundos entre as duas estruturas*

Com esses dados concluímos que não existe diferença significativa na inserção de novos nós na cabeça da lista. A deleção de itens aleatórios da lista sofreu um aumento no tempo de execução de aproximadamente 34% causado pelas operações de xor. Por fim, a inserção de nós em posições aleatórias foi a que sofreu a maior diferença, a xll foi aproximadamente 5700% mais lenta.

Ambas as operações de inserção na cabeça da lista e deleção de um item aleatório da xll possuem a mesma complexidade assintótica de suas contrapartes na dll, porém o tempo da operação de deleção da xll e da dll possuem uma diferença significativa. Podemos concluir que tal diferença de tempo é causada pelas O(n) operações de xor feitas pela função de deleção da xll, pois essa é a única diferença entre as funções. Enquanto isso, a operação de inserção após um nó específico possui complexidade de tempo de execução igual a O(n) na xll, enquanto na dll, tal função é O(1). Vale ressaltar que a função pode ser transformada em O(1) caso seja fornecido o nó alvo e também um de seus vizinhos.

Assim, usando essas três funções diferentes concluímos que, para funções que possuem O(1) operações de xor e mantém a mesma complexidade de tempo, seu tempo de execução não irá diferir muito da implementação comum. Funções que possuem O(n) operações de xor e mantém a mesma complexidade de tempo terão um aumento considerável no tempo de execução. Por fim, as funções que possuem O(n) operações de xor e possuem complexidade de tempo maior que sua contraparte, terão um grande aumento no tempo de execução.

3.6 Conclusão

Neste capítulo foi discutido um tipo diferente de lista duplamente ligada, que para poupar espaço armazena o endereço de ambos seus vizinhos em uma só variável. Foi discutido o impacto na performance que o uso de tal estrutura possa trazer dependendo de quais operações serão mais realizadas na lista.

Embora computadores modernos tenham vastas quantidades de memória disponível em comparação a anos atrás, com a popularização da Internet das coisas, mais uma vez sistemas com pouca memória vem sendo utilizados em escala. Cabe aos desenvolvedores usarem seus conhecimentos para escolher quais algoritmos e quais estruturas serão usados para contornar as limitações desses dispositivos.

Capítulo 4

Bloom Filter

4.1 Introdução ao problema

Devido à grande diferença de velocidade de leitura e escrita entre a memória RAM e a memória de disco (tanto disco rígidos como SSDs), é comum programadores evitarem ao máximo operações que acessam a memória de disco. Embora, em geral, a quantidade de memória RAM disponível nos dispositivos mais novos venha aumentando, sua utilização para armazenar conjuntos inteiros de dados é limitada por conjuntos de dados grandes ou situações em que os dados necessitam ser persistidos, pois RAM se trata de uma memória volátil.

Neste capítulo iremos explorar uma estrutura de dados chamada de Bloom filter, que tem como função verificar rapidamente, e utilizando pouca memória, se um dado elemento não pertence a um conjunto de dados. Embora muitos tipos de Bloom filters tenham sido criados ao longo dos anos, iremos focar no original que foi concebido por Bloom, 1970.

Bloom filters são encontrados em diversos programas modernos. Alguns exemplos são: Sistemas gerenciadores de bancos de dados como o PostgreSQL e Cassandra os utilizam para aumentar a velocidade das consultas ao banco; o navegador Chromium usa a estrutura para filtrar URLs maliciosas; a *block chain* Ethereum utiliza bloom filters para diminuir o tempo de procura por logs; por fim, o website Medium utiliza a estrutura no seu sistema de recomendação de artigos.

4.2 Funções de hash

Iremos usar uma definição simples para uma função de hash $h(U)$. Esta será qualquer função $h : U \longrightarrow \mathbb{N}_m$, e sem perda de generalidade, iremos assumir que U é igual a $\mathbb{N}$. Como h é uma função, um elemento x sempre será mapeado para o mesmo valor, isto é, se $h(x) = y$ então $h(x) = z \longrightarrow y = z$. No entanto, como h não é injetora, temos as chamadas colisões, nome dado ao evento em que $h(x) = y, h(z) = y \, e \, x \neq z$. Além disso, o valor calculado pela função de hash de um certo elemento é comumente chamado de hash do elemento ou valor de hash.

Existem muitas funções de hash para sequências de caracteres (strings). Um exemplo simples é a seguinte função hash:

$$h(s) = \sum_{i=0}^{n-1} s[i] * p^i \ mod \ m, \ p \in \mathbb{N}, m \in \mathbb{N}$$

Existem funções de hash mais complexas, por exemplo as funções de hash criptográficas SHA256 e MD5, como são usadas para fins criptográficos, foram criadas para diminuir ao máximo a possibilidade de que seu valor seja invertido facilmente. No entanto, pela definição dada anteriormente uma simples função $f(x) = x + c \ mod \ m$ também se qualifica como uma função de hash.

4.3 Manipulação de bits

Nas linguagens de programação mais comuns como C, Java, Python, Go etc, não existe uma maneira direta de acessar o valor de um bit, ou de atribuir o valor de 1 ou 0 para um bit. Todavia, em geral elas suportam operações lógicas entre bits, comumente chamadas de operações bit a bit, que possibilitam realizar tais feitos. Todas as representações binárias a seguir são representações sem sinal.

Para descobrir o valor de um único bit de um byte x, basta usar a operação AND bit a bit (geralmente representada por um &) com um valor y, onde todos os bits de y são 0, menos o bit que se deseja descobrir o valor em x. Por exemplo, para descobrir o valor do terceiro bit, basta realizar a operação $x \ \& \ 4$ (4 é representado por 100), se o resultado dessa operação for 0, então o quarto bit de x é 0, caso o resultado for diferente de 0, então o bit tem valor 1.

Para atribuir o valor 1 a um certo bit de um byte x, basta realizar a operação OR bit a bit (geralmente representada por um |) com um valor y, onde todos os bits de y são iguais a 0, menos o bit que se deseja atribuir o valor de 1. Por exemplo, para atribuir o valor de 1 ao quarto bit de x, basta realizar a operação $x = x \mid 8$ (8 é representado por 1000 em binário). Para atribuir o valor de 0 a um certo bit, é necessário fazer o oposto, se usa a operação AND bit a bit, todos os valores de y são iguais a 1, menos o bit que se deseja ter o valor alterado. Por exemplo, $x = x \ \& \ 2$ (2 é representado por 0...010 em binário) irá atribuir o valor de 0 ao penúltimo bit de x.

Para multiplicar um número inteiro por 2^x, basta fazer um deslocamento de bits para a direita ($\ll x$). Para realizar a divisão inteira por 2^x, basta aplicar um deslocamento de bits para a esquerda ($\gg x$).

Por fim, para números inteiros positivos x e para todo n potência de 2, a operação $x \ mod \ n$ é equivalente a operação AND bit a bit por $n - 1$ ($x \ \& \ n - 1$). Isso é possível, pois como n é potência de 2, $n - 1$ irá possuir representação em bits igual a 0...01...1, assim, quando é feita a operação AND bit a bit com um desses números, o valor resultante será igual ao valor dos últimos bits do número original.

4.4 A estrutura

O bloom filter é uma estrutura de dados probabilística, no caso de bloom filters tradicionais, isso quer dizer que não é possível saber se um item está presente no conjunto que a estrutura representa. Existe somente uma probabilidade de que o item esteja no conjunto. Porém, é possível saber com 100% de certeza se um item não está presente.

Um bloom filter tradicional é composto de duas partes, uma estrutura de armazenamento que permite o acesso individual de bits e um grupo de k funções de hash. A estrutura possui duas funções cruciais, a função que adiciona um elemento e a função que determina se um elemento não está presente. Iremos analisar a implementação encontrada no programa PostgreSQL feita na linguagem C.

```
1  struct bloom_filter
2  {
3     /* K hash functions are used, seeded by caller's seed */
4     int k_hash_funcs;
5     uint64 seed;
6     /* m is bitset size, in bits. Must be a power of two <= 2^32. */
7     uint64 m;
8     unsigned char bitset[FLEXIBLE_ARRAY_MEMBER];
9  };
```

Programa 4.1: *Struct usada na implementação, label=bloom:struct*

```
1  void
2  bloom_add_element(bloom_filter *filter, unsigned char *elem, size_t len)
3  {
4     uint32 hashes[MAX_HASH_FUNCS];
5     int i;
6
7     k_hashes(filter, hashes, elem, len);
8
9     /* Map a bit-wise address to a byte-wise address + bit offset */
10    for (i = 0; i < filter->k_hash_funcs; i++)
11    {
12       filter->bitset[hashes[i] >> 3] |= 1 << (hashes[i] & 7);
13    }
14 }
```

Programa 4.2: *Função de inserção*

Para adicionar um item na estrutura, primeiro é calculado o hash do elemento a ser inserido com k funções de hash diferentes, cada um desses k valores representa uma posição do vetor de bits, então tais posições terão o valor de 1 atribuído a elas. Na linha 7 de 4.2, o valor das k funções de hash são calculados e armazenados em *hashes*. Na linha 12 de 4.2, as posições definidas pelos valores de hash tem o valor de 1 atribuído a elas, para isso são dados os seguintes passos: o valor das funções de hash, *hashes[i]*, representa o bit da estrutura que deve ter o valor de 1 atribuído a ela. Para transformar a posição em bits para uma posição do vetor *bitset* é preciso fazer a divisão pelo tamanho em bits de *unsigned char* (8 bits). Já para descobrir qual dos 8 bits de *bitset[hashes[i] » 3]* irá armazenar a informação, é feita uma operação módulo 8 no valor de hash (*hashes[i] & 7*). Então, é efetuado um deslocamento de bits para a esquerda, textit1 « (hashes[i] & 7), tal operação

resulta em um número que possui o valor de 1 no bit desejado e o valor de 0 nos bits restantes. Por fim, é usada a operação OR bit a bit para realizar a atribuição do valor em *bitset[hashes[i] » 3]*.

```
1   bool
2   bloom_lacks_element(bloom_filter *filter, unsigned char *elem, size_t len)
3   {
4      uint32 hashes[MAX_HASH_FUNCS];
5      int i;
6
7      k_hashes(filter, hashes, elem, len);
8
9      /* Map a bit-wise address to a byte-wise address + bit offset */
10     for (i = 0; i < filter->k_hash_funcs; i++)
11     {
12        if (!(filter->bitset[hashes[i] >> 3] & (1 << (hashes[i] & 7))))
13           return true;
14     }
15
16     return false;
17  }
```

Programa 4.3: *Função para detectar se um elemento não se encontra na estrutura*

A função para verificar se um elemento não se encontra na lista é semelhante à função para inserir um elemento. São calculados os k valores de hash do elemento, então verifica se um dos bits que os k valores representam possui valor igual a 0, pois se tal elemento tivesse sido inserido na estrutura, esses bits deveriam ter valor 1. A grande diferença do código de 4.2 e 4.3 é que na linha 12 de 4.3 é usado um AND bit a bit (&) e um *if* para verificar se o valor do bit é igual a 1.

Foi explicado como é possível detectar se um elemento não se encontra na estrutura, resta entender porque não é possível afirmar que um elemento se encontra na lista. Esse fato é devido às colisões das funções de hash e também, pois, na estrutura, só é armazenado se um item com o valor hash específico foi ou não inserido. Por isso torna-se impossível saber qual dos muitos itens que possuem o mesmo valor de hash foi inserido. Por um motivo semelhante não é possível remover itens da estrutura, pois embora seja possível calcular todos os k bits que representam o item na estrutura. É impossível identificar se algum outro elemento inserido na estrutura não compartilha algum desses k bits. Vale ressaltar que, com certas alterações na estrutura, seria possível implementar uma função de deleção, como foi mostrado por Rothenberg *et al.*, 2010.

Nesse sentido, duas dúvidas naturais são como calcular a probabilidade de um falso positivo (quando um item que não está na lista retornaria que ele se encontra na lista) e como a quantidade de funções de hash usadas e o tamanho do vetor usado impactam tal probabilidade. Essas perguntas foram inicialmente respondidas em Mullin, 1983. Primeiro, duas suposições são feitas, todas as equações de hash usadas possuem uma distribuição uniforme e as funções são independentes umas das outras. Com isso temos que: a probabilidade de que um bit qualquer não ser usado na inserção de um novo item é igual a

$$\left(1 - \frac{1}{m}\right)^{k}$$

onde m é o tamanho em bits do vetor de bits e k é o número de funções de hash usadas. A probabilidade de que um bit não seja usado após n inserções é

$$\left(1 - \frac{1}{m}\right)^{kn}$$

Logo, a probabilidade, p_{set}, de que um bit seja usado após a inserção de n itens é

$$p_{set} = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

finalmente, a probabilidade de que k bits tenham sido atribuídos previamente é igual a

$$p_{false} = p_{set}^{k} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{4.1}$$

A equação (4.1), geralmente referida como “equação clássica”, é usada na implementação encontrada no PostgreSQL para calcular o número de funções de hash que serão usadas. O espaço de memória máximo que deve ser usado (m) e uma estimativa de quantos elementos serão inseridos na estrutura (n) são fornecidos na criação da estrutura. Para calcular k, a equação (4.1) é aproximada para

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

então o número de funções de hash a serem usadas será igual ao número inteiro k que minimiza p_{false}, que é equivalente a $\frac{m}{n} \ln 2$. Usando o número calculado, a porcentagem de falsos positivos estará entre 1% e 2%.

4.5 Simulação

Para ajudar na compreensão da estrutura, será feita uma ilustração de seu funcionamento. Nessa exemplificação, será usada uma estrutura de dados fictícia que nos permite acessar bits individualmente. Serão usadas 3 funções de hash.

Inicialmente o bloom filter será igual a:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Iremos inserir a palavra "bcc"(os valores de hash dessa palavra serão 3, 7 e 14). Após a inserção o bloom filter estará no seguinte estado:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0

Após inserir a palavra "algoritmo"(que possui valores de hash iguais a 0, 6 e 8), o bloom filter estará no seguinte estado:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	1	0	0	1	1	1	0	0	0	0	0	1	0

Com a inserção da palavra "bloom"(que possui valores de hash iguais a 2, 11 e 14), o bloom filter estará no seguinte estado:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	1	1	1	0	1	0	0	0	1	0

Como a palavra “filter” (que possui valores de hash iguais a 2, 6 e 13) não se encontra na estrutura, a função de verificação se a palvra não se encontrada na estrutura ira retornar verdadeiro, pois a posição de índice 13 possui valor 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	1	1	1	0	1	0	0	0	1	0

Já a verificação de se a palavra “bcc” (que possui valores de hash iguais a 3, 7 e 14) não se encontra na estrutura irá retornar falso, pois todos os índices (3, 7 e 14) possuem valor igual a 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	1	1	1	0	1	0	0	0	1	0

Enquanto que a verificação da palavra “monografia” (que possui valores de hash iguais a 0, 6 e 14) irá retornar falso, pois todos os índices (0, 6 e 14) possuem valor igual a 1. Claro que tal palavra não está presente na estrutura, porém devido às colisões é impossível determinar isso.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	1	1	1	0	1	0	0	0	1	0

4.6 Conclusão

Neste capítulo foi feita uma análise sobre o Bloom filter, uma estrutura de dados probabilística criada nos anos 1970, que é usada para responder rapidamente, sem usar muita memória, se um elemento não pertence a um conjunto de dados. Houve também uma breve explicação de funções de *hashing* e de manipulação de bits.

Espera-se que com este capítulo tenha sido introduzido ao leitor uma classe nova de estruturas de dados, uma estrutura usada atualmente por muitos projetos de código aberto (PostgreSQL, Chromium, Cassandra etc). Também foi dada uma explicação de como acessar e atribuir valores a bits individuais.

Capítulo 5

B+Tree

5.1 Introdução ao problema

Sistemas de arquivos e sistemas gerenciadores de bancos de dados lidam com grandes quantidades de dados. As operações de busca, adição e remoção desses sistemas devem ser rápidas, para que os usuários tenham uma boa experiência de uso. Como para a maioria dos casos de uso desses sistemas é inviável armazenar todos os dados em memória RAM, foi necessário criar uma maneira mais eficiente, em detrimento da simples ideia de usar operações lineares em disco.

Um primeiro passo para tornar tais sistemas mais eficientes é o de usar paginação, agrupar os dados em montantes que caibam na memória RAM e marcar cada página com um índice. Assim, por exemplo, 10000 itens podem ser divididos em páginas de 100 itens cada, sendo que os 100 índices que representam as páginas podem ser facilmente armazenados em memória RAM. Então, basta procurar qual página os dados se encontram usando os índices, para depois fazer uma procura pelos dados em si na página.

Ainda podemos melhorar mais esse sistema. Em vez de armazenar os índices em uma estrutura linear como um vetor ou como uma lista, podemos usar árvores de busca binária autobalanceadas, estruturas que possuem tempos de busca, inserção e remoção logarítmicos. Neste capítulo, iremos explorar uma árvore muito utilizada para tais aplicações, a Árvore B e apontar uma de suas modificações muito utilizadas, a Árvore B+.

5.2 Árvores balanceadas

Árvores de busca binárias (BST) aparentam prometer operações de inserção, deleção e procura em tempo O(lg(n)) (n representando o números de elementos na árvore), porém, na realidade tais operações são O(n). O motivo para isso é simples, tais operações são todas dependentes da altura da árvore, ou seja, são O(h), onde h é a altura da árvore, e a altura de BSTs variam de lg(n) até n.

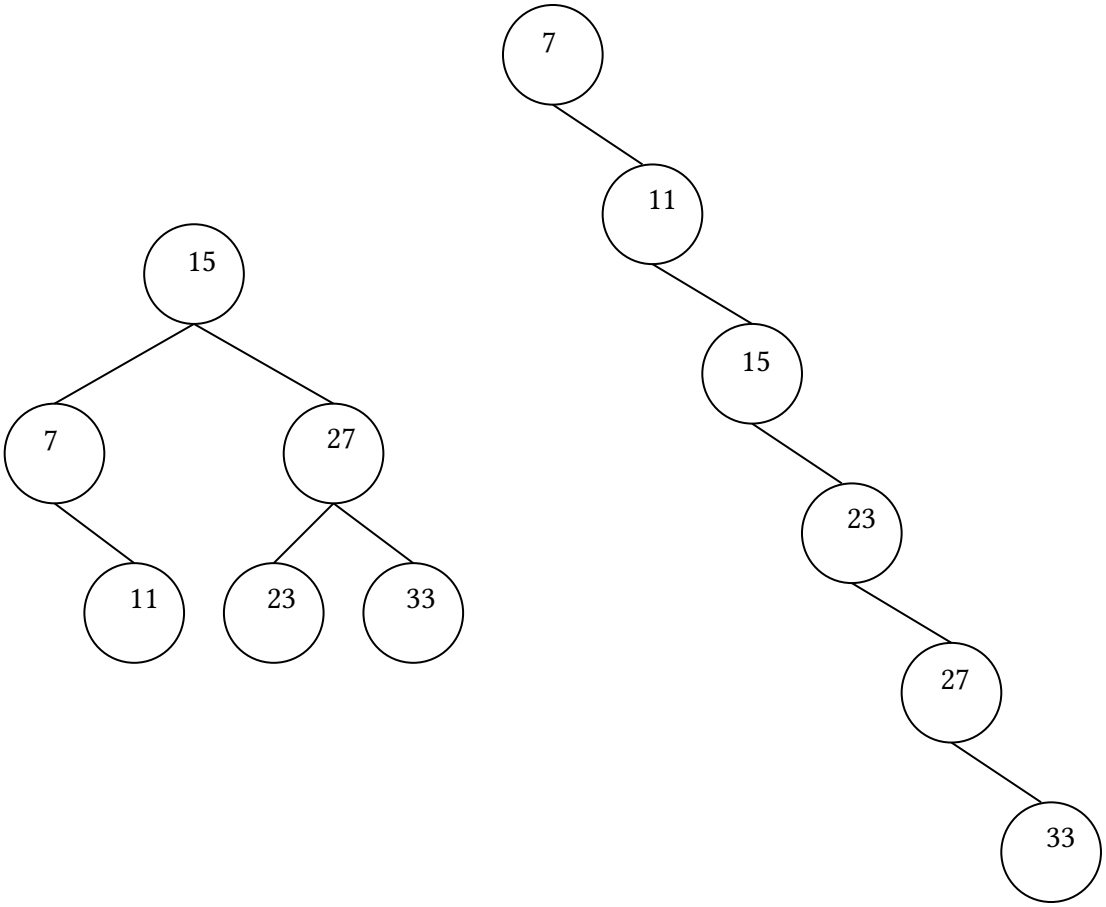

Figura 5.1: *Duas BSTs com os mesmos elementos*

Como pode ser visto na figura 5.1, temos duas BSTs com os mesmos elementos, porém com alturas diferentes, uma possui altura igual a lg(n) e a outra igual a n. Isso se deve ao fato de que os itens dessas árvores foram inseridos em ordens diferentes. Somente árvores que possuem altura próxima de lg(n), chamadas de árvores balanceadas, são de nosso interesse e as iremos definir como todas as árvores que possuam altura O(lg(n)).

Por fim, iremos introduzir o conceito de árvores autobalanceadas, que são árvores balanceadas que após qualquer operação de inserção ou remoção, permanecem balanceadas. Por exemplo, árvores rubro-negras, árvores AVL e árvores B são todas árvores autobalanceadas.

5.3 Árvore B

Árvores B (B-Trees) são um tipo de árvore de busca autobalanceada, criada por Beyer e McCreight, 1972. Pra uma árvore ser classificada como uma árvore B de classe k, as seguintes propriedades devem ser verdadeiras:

i O tamanho do caminho da raiz da árvore para qualquer folha da árvore é o mesmo.

ii Todos os nós, com exceção da raiz e das folhas, possuem no mínimo $k + 1$ filhos. Se a raiz não for uma folha ela terá no mínimo 2 filhos.

iii Todo nó tem no máximo 2k + 1 filhos.

A inserção de novas chaves em uma B-tree é feita nas folhas da árvore, assim primeiro é preciso fazer uma busca do local onde tal chave se encontraria na árvore. Após identificar em qual folha a nova chave deve ser inserida, existem duas possibilidades. A primeira delas é a de que, caso exista espaço livre na folha, basta inserir a chave na posição apropriada da folha,

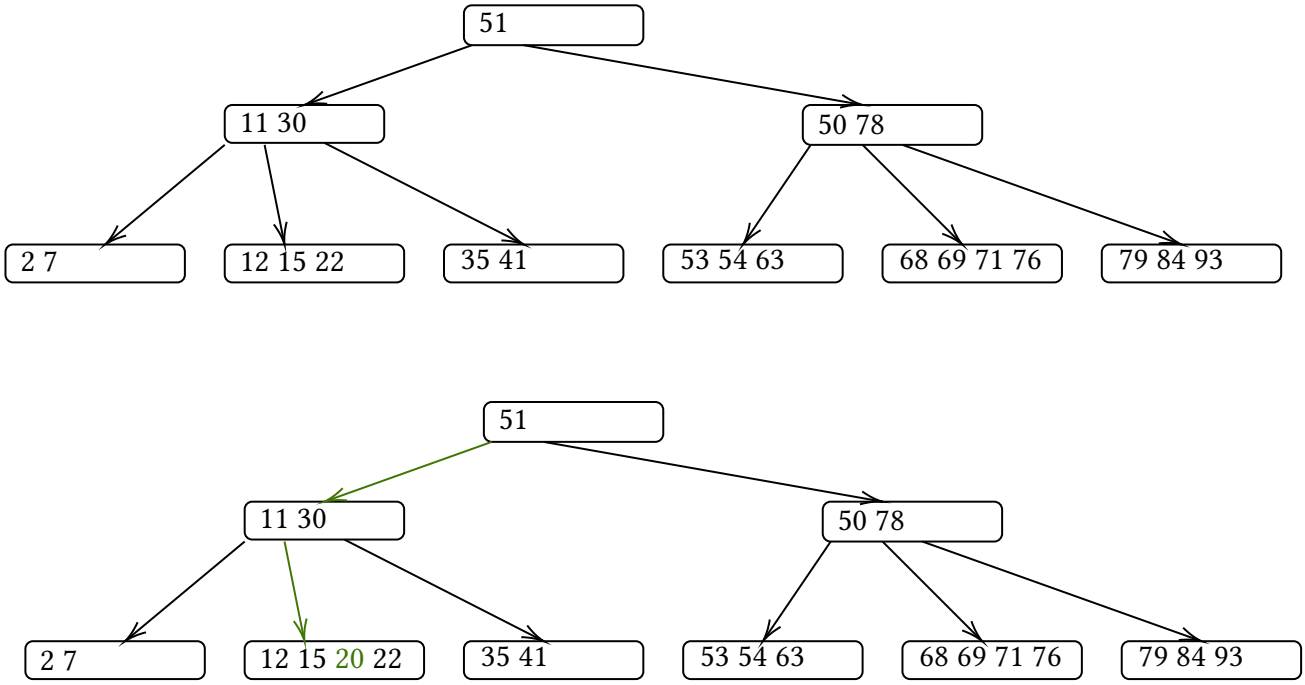

Figura 5.2: *Inserção de um item com chave de valor igual a 20 em uma árvore B de classe 2*

a segunda possibilidade é a de que, caso a folha se encontre cheia, após a inserção da nova chave é necessário rebalancear a árvore. Nesse caso será feito a divisão da folha em duas, com metade das chaves indo para cada nova folha, a chave central será inserida no nó pai das folhas. Tal operação de rebalanceamento se repete recursivamente caso o pai também se encontre cheio.

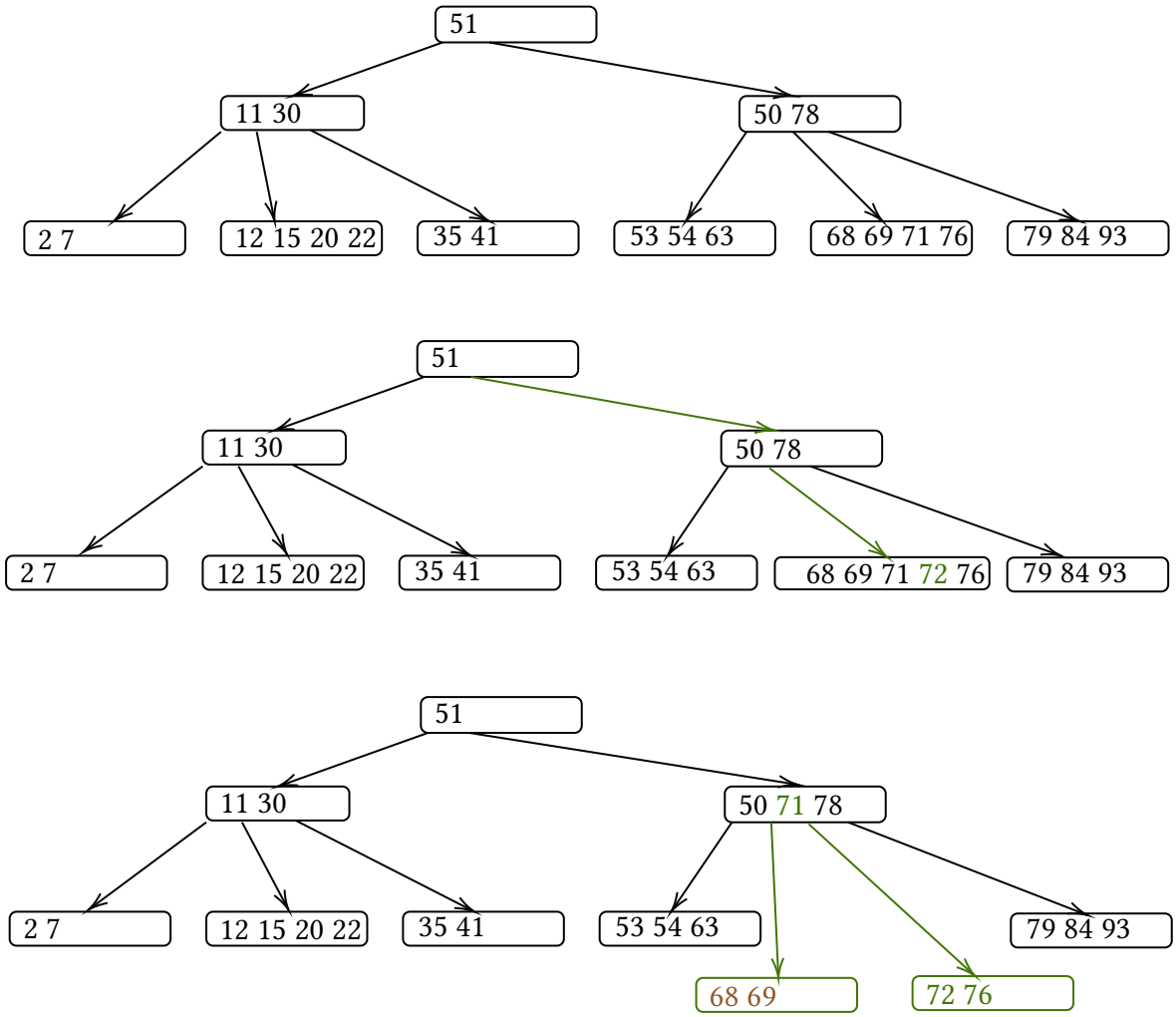

Figura 5.3: *Inserção de um item com chave de valor igual a 72 em uma árvore B de classe 2*

A deleção de chaves também irá focar em operações de folha. Sendo que existem os seguintes casos para a remoção de uma chave:

I. A chave a ser removida se encontra em uma folha.

   i. A folha em que a chave se encontra possui mais de k elementos.

   ii. A folha em que a chave se encontra possui k elementos e um de seus vizinhos possui mais de k chaves.

   iii. A folha em que a chave se encontra possui k elementos e ambos dos seus vizinhos possuem k chaves.

II. A chave a ser removida se encontra em um nó interno.

   i. A folha descendente esquerda do nó interno que possui a maior chave tem mais de k chaves, ou a folha descendente direita do nó interno que possui a menor chave à direita tem mais de k chaves.

   ii. A folha descendente esquerda do nó interno que possui a maior chave tem k chaves e a folha descendente direita do nó interno que possui a menor chave à direita tem k chaves.

Para o caso I.i, basta remover a chave da folha em que ela se encontra, após tal remoção a árvore ainda será uma B-tree válida.

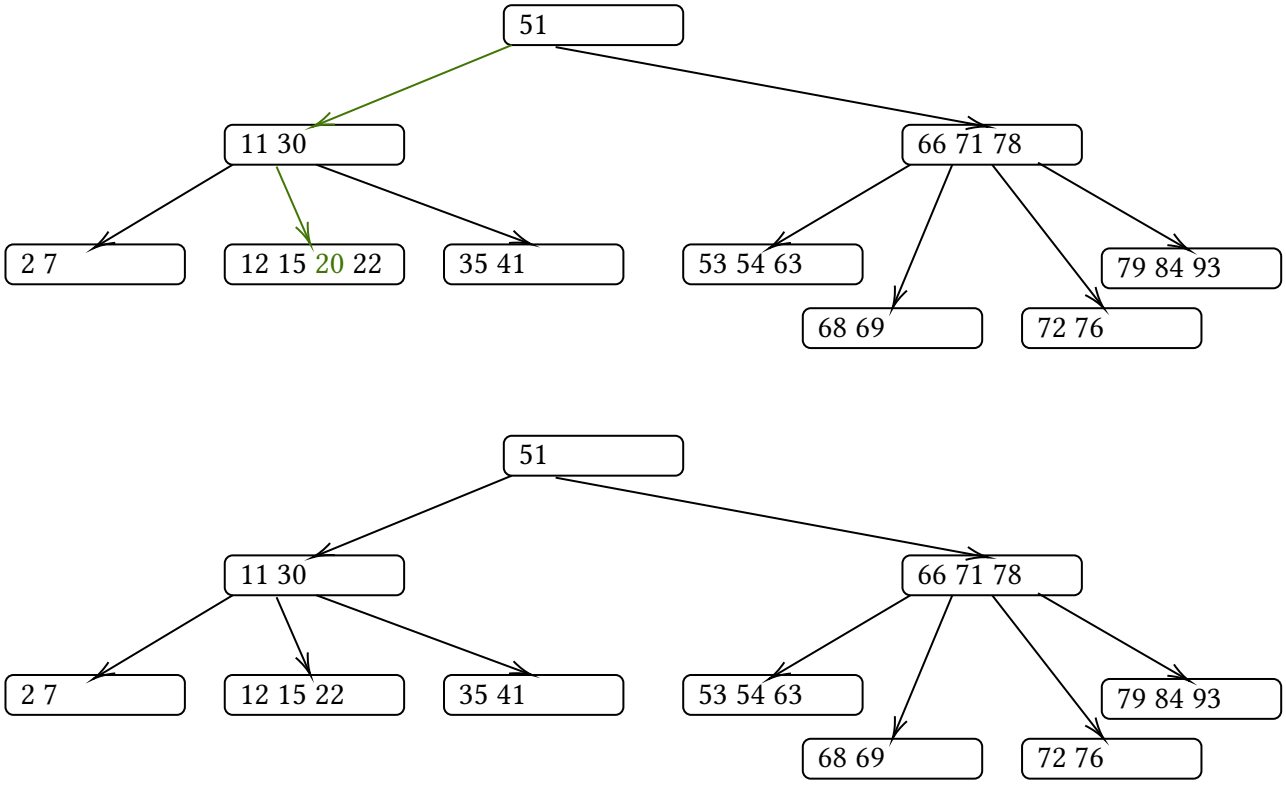

Figura 5.4: *Remoção de item com chave igual a 20 em uma árvore B de classe 2*

Para o caso I.ii, como a folha possui k chaves, somente remover a chave da folha irá quebrar uma das propriedades da árvore B. Como um dos vizinhos possui mais de k chaves,

é possível fazer uma redistribuição simples de chaves. Caso o vizinho esquerdo possua mais de k chaves e ele seja escolhido, o maior elemento do vizinho será realocado para o nó pai, e a chave que pertence ao nó, que divide o vizinho esquerdo e a folha, será realocada para a folha. Caso o vizinho direito seja escolhido o processo será similar, com a mudança de que a menor chave do vizinho será realocada para o nó pai.

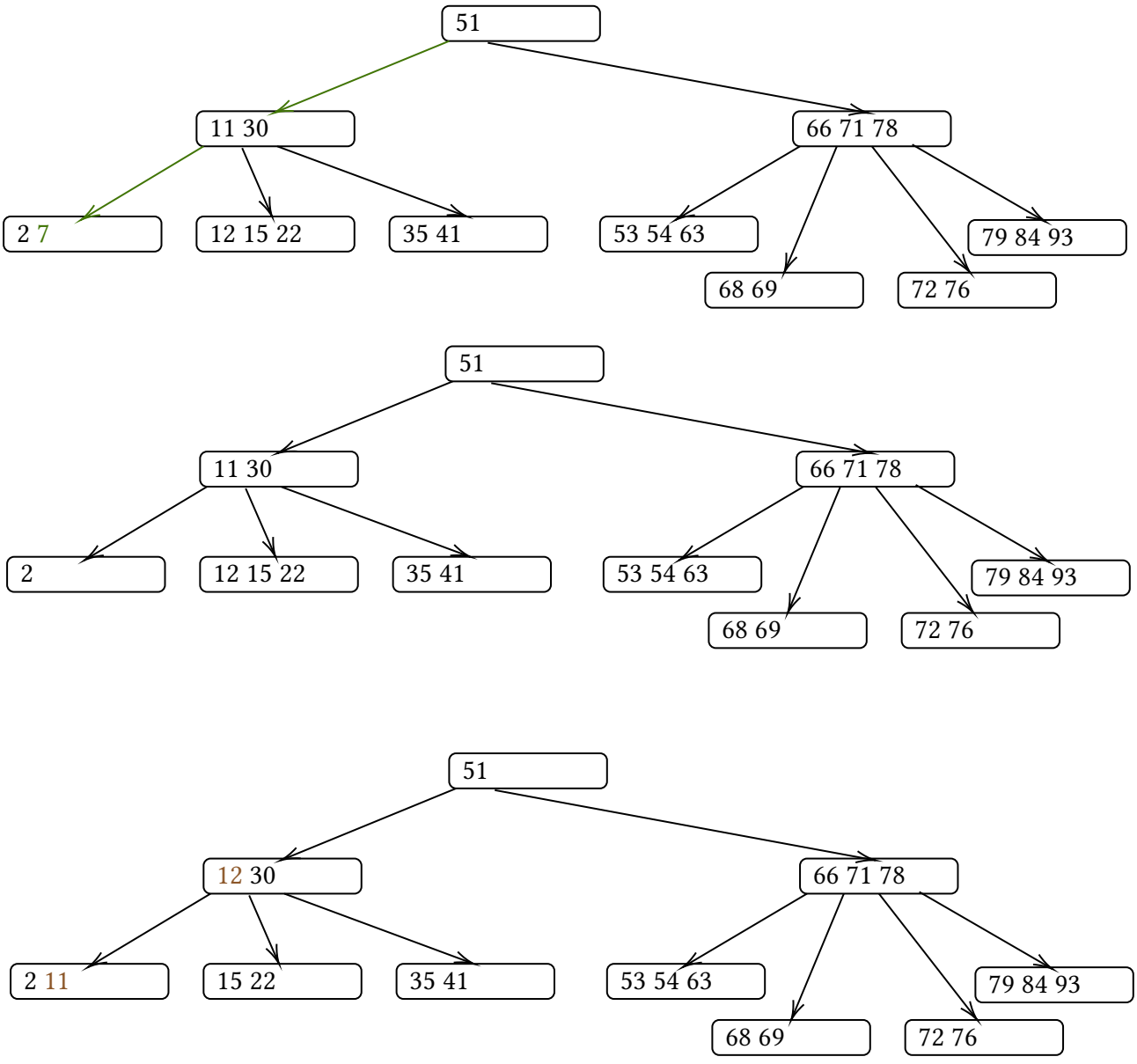

Figura 5.5: *Remoção de item com chave igual a 7 em uma árvore B de classe 2*

No caso I.iii, como ambos os vizinhos da folha possuem somente k chaves, não é possível realizar a operação de redistribuição. Após a remoção da chave da folha, a folha será combinada com um de seus vizinhos e a chave do nó pai que as divide. Caso o nó pai se encontre com menos de k chaves após essa operação, as mesmas operações de distribuição e combinação de nós discutidas anteriormente serão realizadas recursivamente para manter os requisitos de uma B-tree.

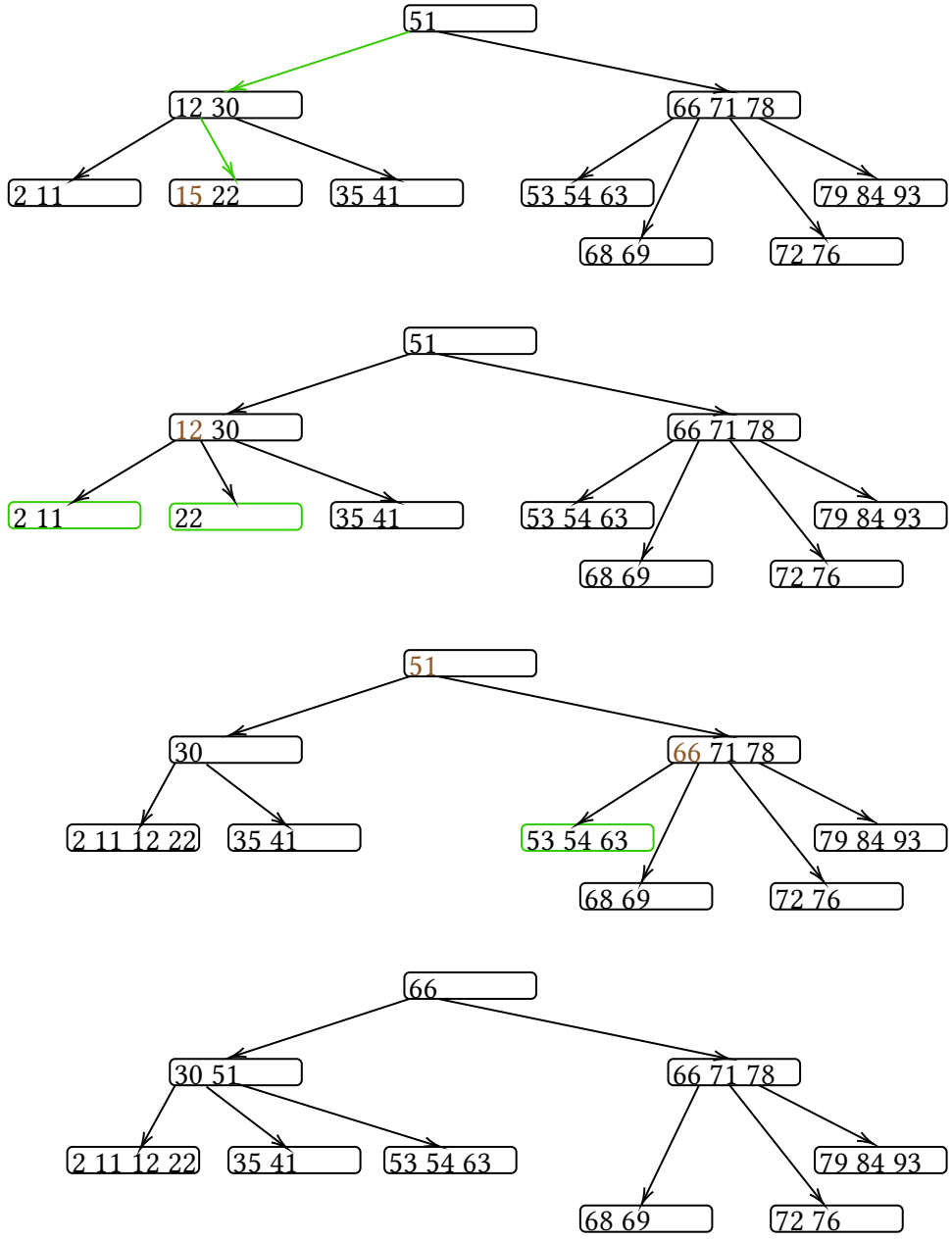

Figura 5.6: *Remoção de item com chave igual a 15 em uma árvore B de classe 2*

No caso II.i, existe uma folha descendente à esquerda ou à direita do nó com mais de k chaves, assim podemos retirar uma chave de tal folha e realoca-lá para a posição da chave que desejamos remover. Se o descendente à esquerda foi escolhido, use a chave com maior valor; caso o descendente à direita for escolhido, use a chave com o menor valor.

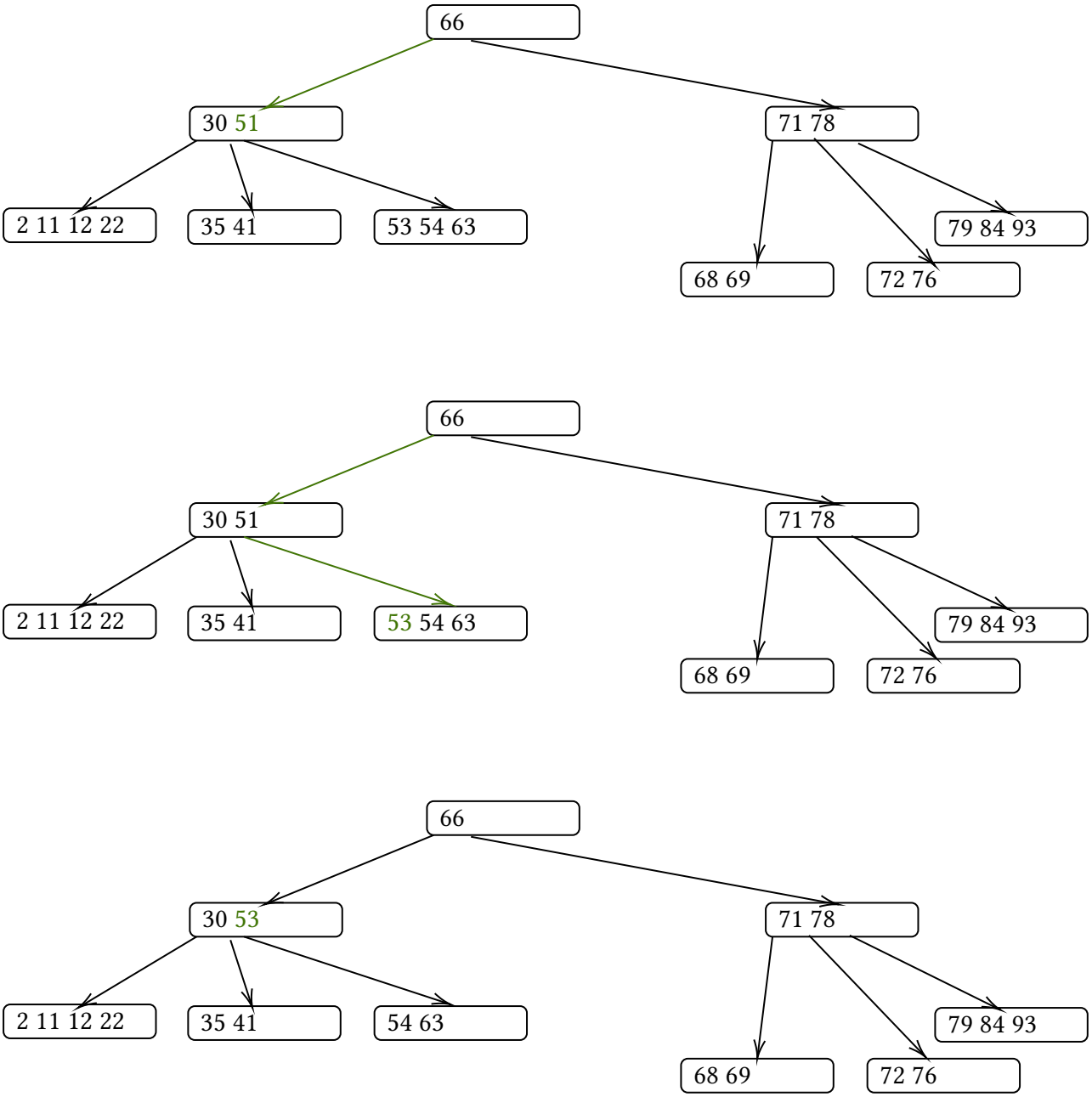

Figura 5.7: *Remoção de item com chave igual a 51 em uma árvore B de classe 2*

Por fim, no caso II.ii, substituir a chave que se deseja remover por uma chave de uma folha esquerda ou direita resultará em uma folha com menos de k chaves, assim as mesmas técnicas de rebalanceamentos usadas em I.ii I.iii devem ser aplicadas.

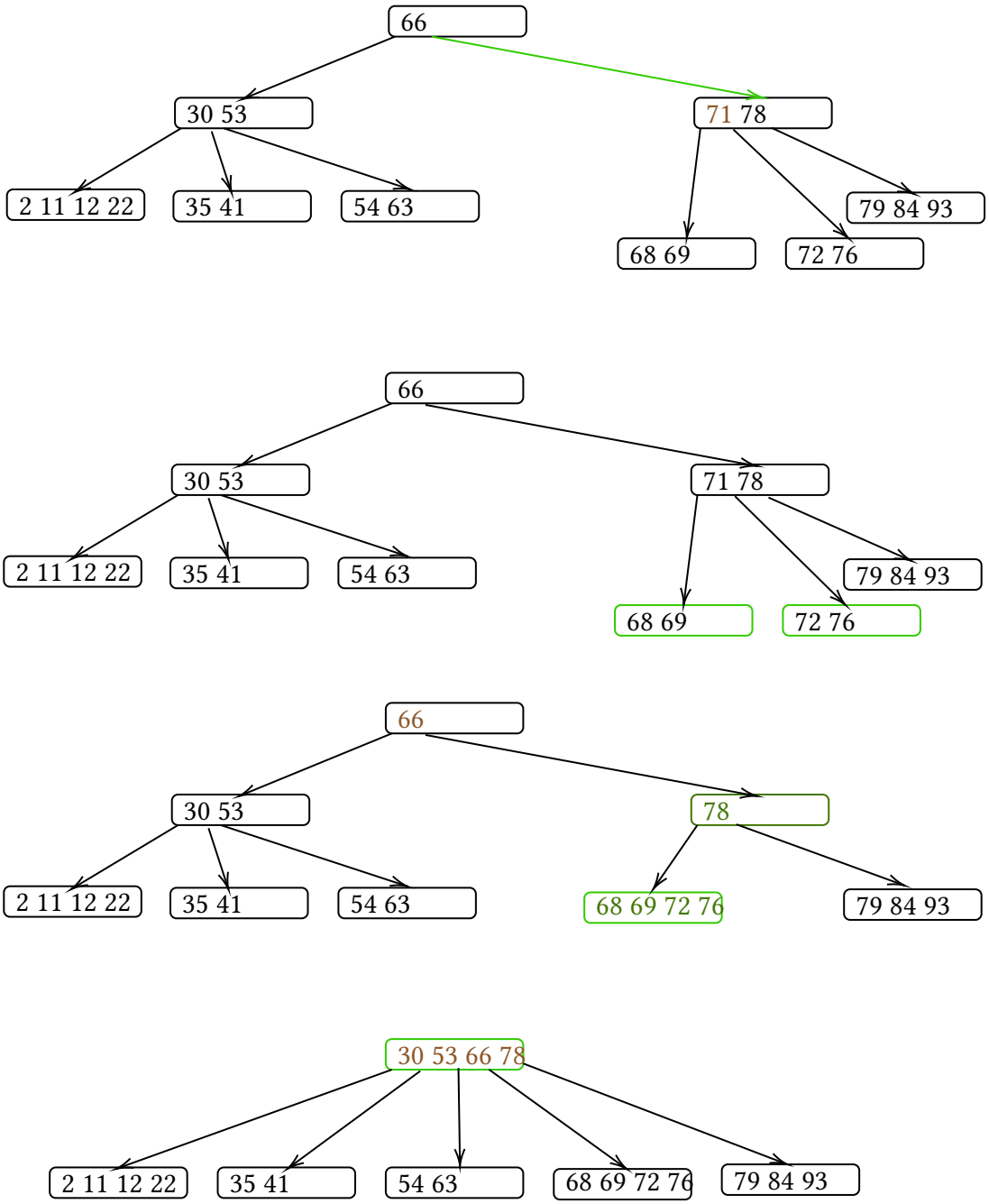

Figura 5.8: *Remoção de item com chave igual a 71 em uma árvore B de classe 2*

5.4 Árvore B+

Existem algumas variantes da Árvore B, uma delas é a Árvore B+, que possui autoria desconhecida, porém aparenta ter sido primeiro mencionada em Knuth, 1973 e melhor detalhada em Comer, 1979.

A árvore B+ é caracterizada por possuir todos os pares chave-valor armazenados nas folhas da árvore, sendo que nos nós internos da árvore somente são armazenadas chaves e ponteiros para os próximos nós. Além disso, opcionalmente as folhas podem formar uma lista ligada. Uma árvore B+ pode ser direitista (as chaves para a esquerda de uma certa chave são menores e as chaves para a direita são maiores ou igual) ou esquerdista (as chaves para a esquerda de uma certa chave são menores ou iguais e as chaves para a direita são maiores).

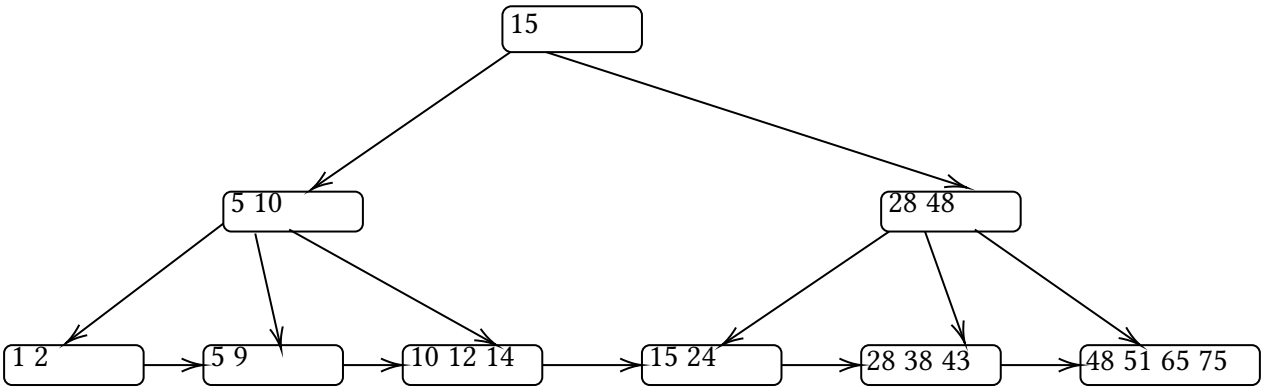

Figura 5.9: *Ilustração de uma Árvore B+ de ordem 2*

A operação de inserção em uma árvore B+ só diverge da operação de inserção em uma árvore B no caso em que a folha cujo novo par chave-valor será inserido está cheia. Nesse caso, as chaves serão divididas em dois e a chave central será inserida na folha esquerda, no caso de uma árvore esquerdista; ou na folha direita, no caso de uma árvore direitista. Além disso, a chave central será propagada para o nó pai.

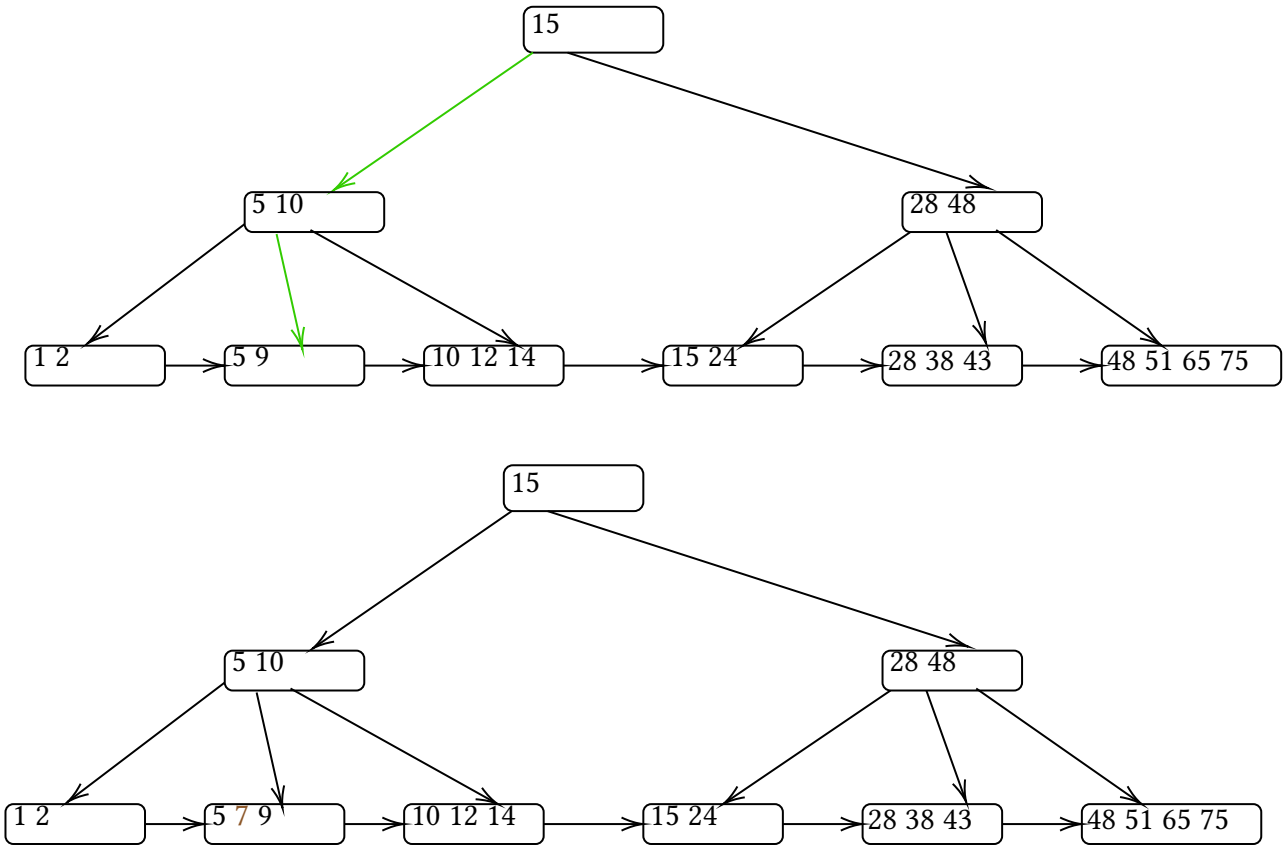

Figura 5.10: *Inserção de item com chave igual a 7 em uma árvore B+ de classe 2*

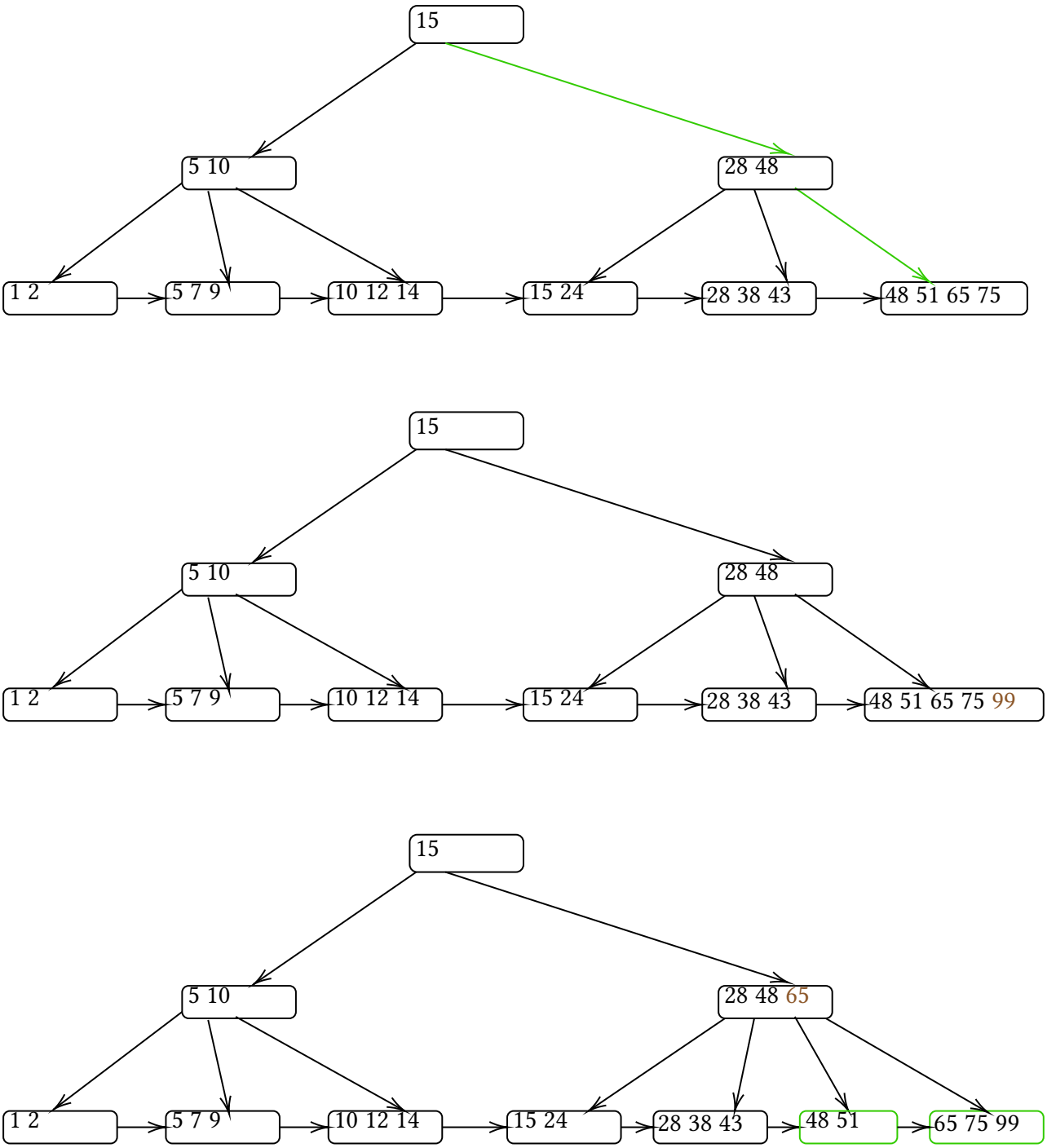

Figura 5.11: *Inserção de item com chave igual a 99 em uma árvore B+ de classe 2*

Uma das vantagens da árvore B+ é a simplificação das operações de deleção, pois, em relação à árvore B, só é preciso cuidar dos casos em que as chaves se encontram nas folhas. A principal mudança é que, após a remoção de um item, deve-se alterar as referências à chave, presentes nos nós antepassados.

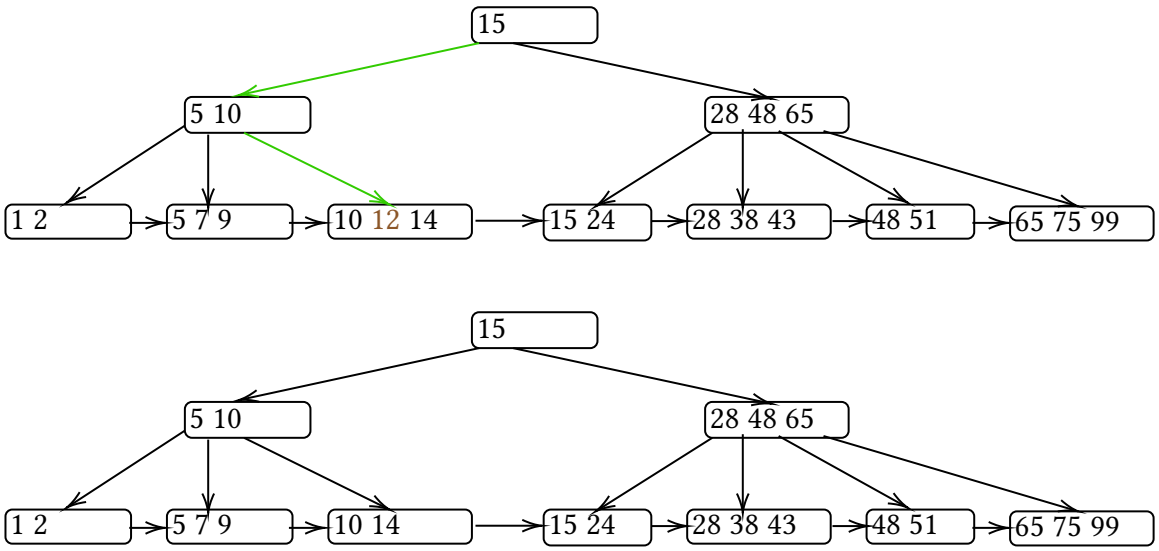

Figura 5.12: *Remoção de item com chave igual a 12 em uma árvore B+ de classe 2*

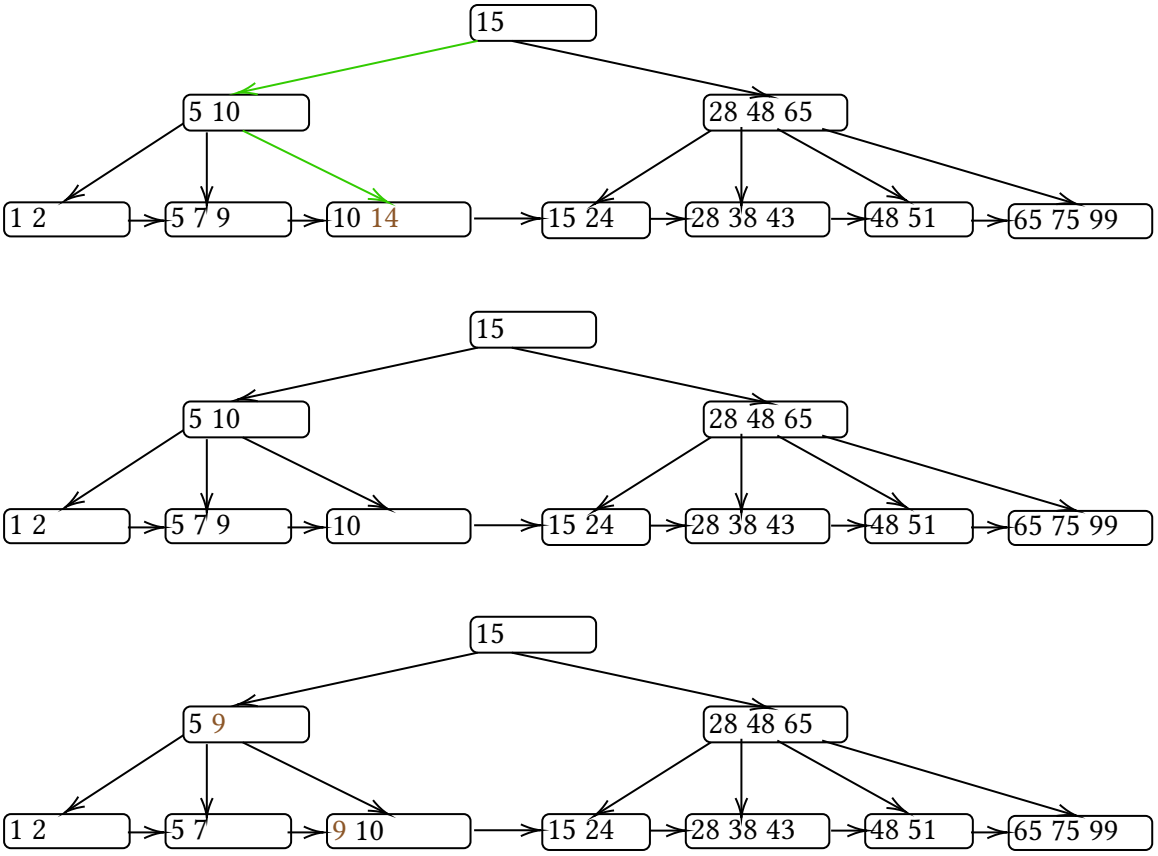

Figura 5.13: *Remoção de item com chave igual a 14 em uma árvore B+ de classe 2*

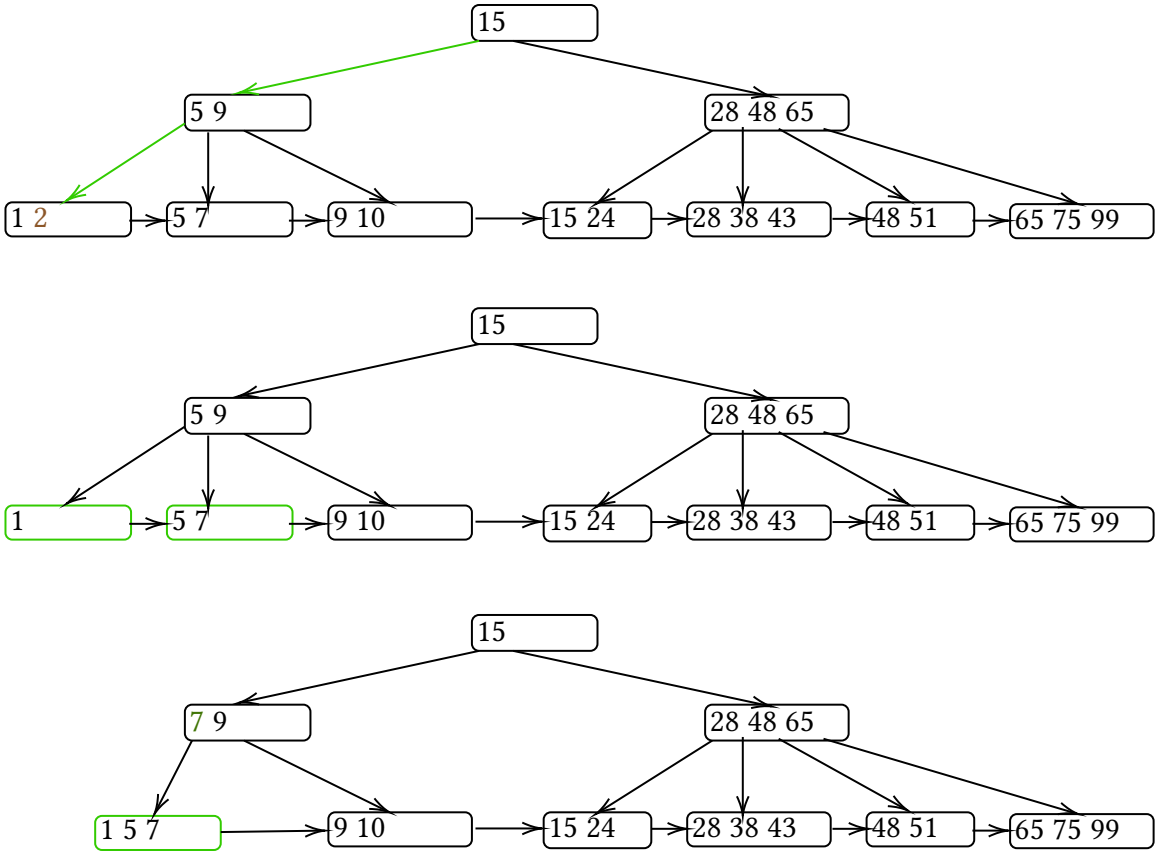

Figura 5.14: *Remoção de item com chave igual a 2 em uma árvore B+ de classe 2*

Um lugar onde se pode encontrar árvores B+ é o *kernel* do Linux. Existem algumas mudanças interessantes da implementação encontrada no *kernel*: Primeiro, as chaves são guardadas em ordem decrescente, para a esquerda de um nó se encontram valores maiores e para a direita se encontram valores menores. As folhas não foram ligadas, pois não se achou uso dessa característica no *kernel*. Para simplificar a árvore, é usada a mesma estrutura de dados para as folhas e para os nós internos, ou seja, em um nó interno, para cada chave existe somente um ponteiro para um descendente.

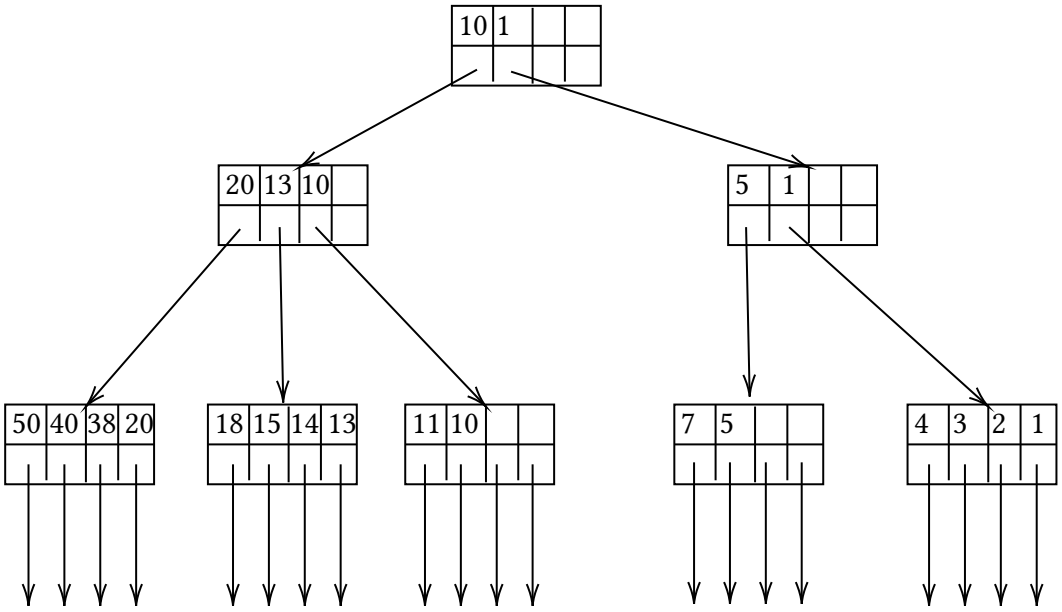

Figura 5.15: *Ilustração da árvore B+ usada pelo Linux*

5.5 Conclusão

Neste capítulo, espera-se que o leitor tenha se familiarizado brevemente com os conceitos de paginação e indexação usados por sistemas de arquivos e SGBDs, e como árvores de busca autobalanceadas são usadas para aumentar a performance. Além disso, se espera que tenha sido introduzido ao leitor uma das muitas variantes de árvores B, a árvore B+.

Capítulo 6

Hierarchical Path-Finding A*

6.1 Introdução ao problema

Calcular a menor distância entre dois vértices de um grafo é um problema comum, e existem muitos algoritmos capazes de resolver esse problema. Entretanto, conforme certas condições são adicionadas ao problema, até mesmo os melhores algoritmos conhecidos por estudantes da graduação não são o suficiente para realizar a tarefa em uma quantidade de tempo adequada ou no espaço de memória disponível.

Iremos analisar dois problemas diferentes: o primeiro é calcular o caminho entre duas cidades distantes, digamos São Paulo até Bogotá ou Cidade do Cabo até Berlim. Embora esses exemplos sejam exagerados, ainda sim deveria ser possível calcular o melhor caminho entre esses pontos, ou pelo menos uma boa aproximação, sem gastar muito tempo e memória. Um outro problema comum encontrado em jogos de estratégia em tempo real é calcular o melhor caminho que uma unidade deve percorrer. Nesse problema geralmente existem várias unidades se movendo até o mesmo destino, bloqueando umas às outras, além do terreno do jogo ser muito mais volátil, com novos caminhos surgindo e sendo bloqueados a cada segundo.

Neste capítulo, iremos explorar como esse problema foi resolvido no jogo *Command & Conquer* de 1995 e iremos introduzir uma modificação do famoso algoritmo A* capaz de resolver ambos os problemas.

6.2 Command & Conquer: Busca de caminho

Command & Conquer é um jogo de estratégia em tempo real (RTS) lançado em 1995. Em um RTS, o jogador controla várias unidades, dando comandos de movimento e de ataque que são todos executados ao mesmo tempo e podem ser cancelados. Por exemplo, após comandar uma unidade a se mover até certa localização, enquanto a unidade realiza o movimento o jogador pode manejar a mesma unidade a fazer outra tarefa. Ao contrário de um jogo por turnos, como o Xadrez, todos os jogadores realizam seus comandos simultaneamente. O jogo foi remasterizado em 2020 e com essa remasterização o código-fonte do jogo foi publicado sob a licença GPLv3.

Pelo fato de que existem várias unidades (agentes) realizando tarefas ao mesmo tempo, e que algumas dessas tarefas envolvem alterar propriedades do mapa (como construir edifícios ou remover árvores) além do fato de que a capacidade computacional dos computadores da época era limitada, os desenvolvedores criaram um algoritmo próprio para *pathfinding*. Será feita uma breve análise desse algoritmo, para melhor entendermos como o problema de busca de caminhos era resolvido em jogos antigos.

Quando um agente precisa se locomover de uma posição x para uma posição y, o caminho que ele deve seguir é calculado da seguinte forma: calcula-se a reta xy e os movimentos do agente seguirão essa reta em direção a y. Caso, durante o caminho se encontre um obstáculo, irá se determinar se existe uma célula na reta após esse ponto que é acessível. Caso não exista tal célula, o destino se encontra em uma célula que não é acessível e o caminho termina no obstáculo. Caso exista uma célula acessível após o obstáculo, serão calculados dois caminhos, um que contorna o obstáculo pela direita e outro que contorna pela esquerda até chegar na célula, o menor desses dois caminhos será escolhido.

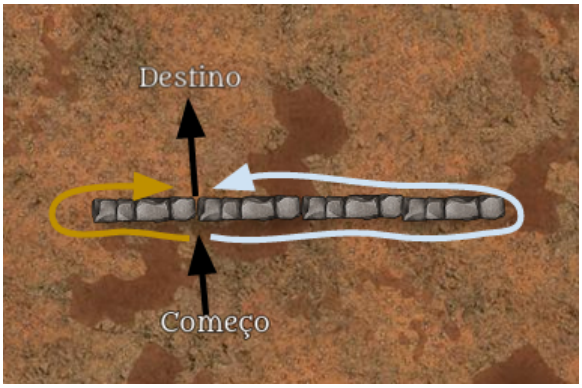

Figura 6.1: *Ilustração de um caminho obstruido*

Por fim, se a célula que se encontra após o obstaćulo for inacessível, ou seja, totalmente cercada por obstáculos, temos o que os desenvolvedores chamaram de uma *doughnut*. Para contornar uma *doughnut*, encontra-se uma nova célula livre de obstáculo dentro da reta xy após o obstáculo original e se fará um caminho até tal célula usando os métodos mencionados anteriormente. Vale notar que se forem encontradas 5 *doughnuts* seguidas, o algoritmo irá terminar devolvendo o caminho de x até o primeiro obstáculo. Não há nenhum comentário no código do porquê isso ocorre, só podemos especular que os desenvolvedores decidiram que encontrar 5 ou mais *doughnuts* consecutivas seria algo raro ou impossível, logo tal caso não precisaria ser tratado. Para acelerar o cálculo do caminho, somente os primeiros 9 passos do caminho são calculados, deixando o resto do caminho para ser calculado posteriormente.

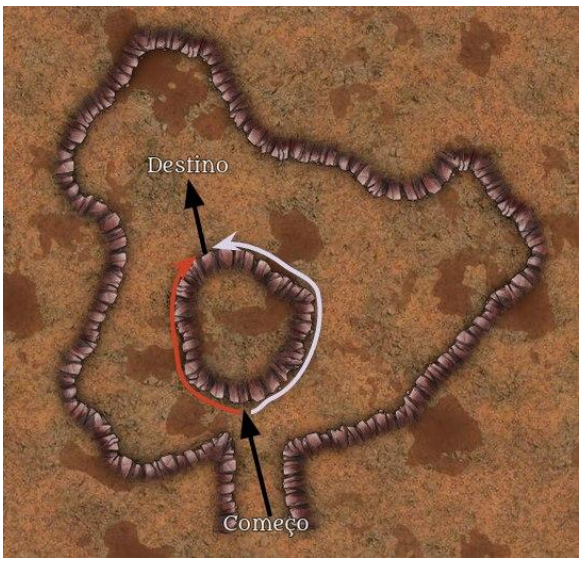

Figura 6.2: *Ilustração de uma doughnut*

Com essa breve explicação do algoritmo usado podemos tirar algumas conclusões sobre métodos de busca de caminhos usados em jogos RTS: não existe uma necessidade de se calcular o caminho exato, uma aproximação basta; casos raros não necessariamente precisam ser tratados; calcular o caminho em partes é uma estratégia valiosa; e por fim, um método aparentemente simples, como seguir uma reta, pode gerar um algoritmo complexo.

6.3 O algoritmo

Um algoritmo muito usado para a busca de caminhos é o A*, que possui complexidade de tempo O(E + V), onde E representa o número de arestas do grafo e V o número de vértices. Mesmo o A* sendo um ótimo algoritmo, possuindo o melhor tempo assintótico possível, para grafos suficientemente grandes ou quando muitos caminhos devem ser calculados simultaneamente, ele pode não ser suficientemente rápido para calcular o resultado para alguma dessas aplicações.

Para contornar os problemas mencionados, com um foco em jogos digitais, em Botea *et al.*, 2004 foi introduzido o Hierarchical Path-Finding A* (HPA*). A principal ideia do HPA* é usar pré-processamento para melhorar a performance do A*. Tal pré-processamento consiste na criação de um grafo G', uma abstração do grafo original G. O grafo G' é formado por agrupamentos (*clusters*) dos vértices de G.

Primeiro é necessário criar o grafo G', a escolha de algoritmo para isso é livre e irá variar dependendo do contexto em que o HPA* será aplicado. Com o grafo G' em mãos, o algoritmo irá encontrar o caminho entre o vértice de origem (S) e o vértice de destino (E) da seguinte forma: primeiro S e E são inseridos no grafo G', os conectando aos agrupamentos aos quais eles pertencem com uma aresta de peso 0. É realizado um A* nesse novo grafo para encontrar o caminho de S a E, com isso, possuímos uma lista de todos os agrupamentos que devem ser visitados no caminho de S para E no grafo G. Por fim, é calculado o caminho de S

a E no grafo original usando o A*, porém as entradas/saídas dos agrupamentos encontrados anteriormente são usadas como pontos intermediários, assim é possível fazer o cálculo de um agrupamento por vez.

Para acelerar o algoritmo é possível fazer outro pré-processamento, calcular o melhor caminho de todos os vértices de entrada/saída de um agrupamento para as outras entradas/saídas desse agrupamento. É também possível usar várias camadas de grafos abstratos, por exemplo o grafo das ruas de uma cidade pode ser abstraído para os bairros dessa cidade e tal grafo pode ser abstraído usando as zonas da cidade.

O algoritmo não necessariamente irá calcular o melhor caminho, porém os resultados experimentais dos pesquisadores mostraram que a diferença entre os tamanhos do caminho ótimo e do caminho gerado pelo HPA* era em torno de 10%.

6.4 Simulação

Para a demonstração do HPA*, será usado um grafo pequeno, que não representa bem o caso de uso real do algoritmo, porém usar um grafo de tal tamanho permite uma melhor ilustração do algoritmo.

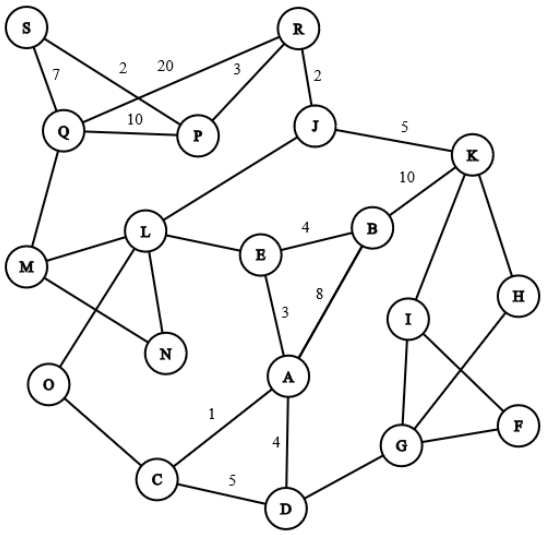

Figura 6.3: *Grafo G*

Iremos calcular o caminho de D a S, para isso primeiro será calculado o caminho de D a S no seguinte grafo G':

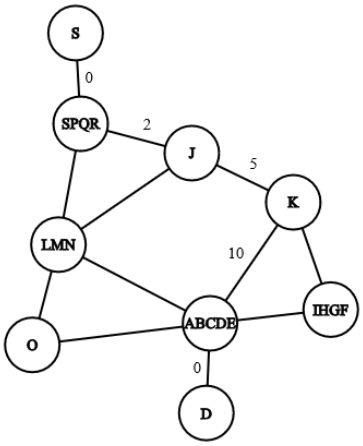

Figura 6.4: *Grafo G'*

Assuma que, os valores das arestas são tais que, o caminho calculado pelo A* será: [D, ABCDE, K, J, PQRS, S]

Tal lista representa os *clusters* que irão ser visitados pelo caminho de D a S. Então serão calculados os caminhos dentro de cada um desses *clusters* até sua fronteira com o próximo *cluster* da lista, ou seja, serão calculados os caminhos: D até B, K até K, J até J e por fim R até S.

De tal forma, o caminho completo de D até S será: D -> A -> E -> B -> K -> J -> R -> P -> S.

Vale ressaltar que, o caminho de todas as entradas de um *cluster* até todas as saídas deste *cluster* já podem ter sido pré-calculadas. Além disso, o próprio grafo G mostrado pode ser somente um grafo intermediário do HPA*,

6.5 Conclusão

Neste capítulo, foram abordados algoritmos de busca de caminho em grafos. Foi feita uma breve análise do algoritmo usado no jogo *Command & Conquer* para cálculo de caminhos, um algoritmo que tenta executar a ideia simples de seguir uma linha reta até o destino. Tal ideia simples resulta em um algoritmo que troca um resultado exato por agilidade no cálculo de um caminho. Vale ressaltar que o código resultante é repleto de casos especiais e repetição de código. Também foi introduzido o HPA*, uma versão modificada do A* criada com aplicação em jogos como foco de pesquisa. O HPA* utiliza uma quantidade arbitrária de representações diferentes do grafo original para realizar o cálculo de caminho de forma mais rápida que o A* tradicional, porém retornando um caminho não necessariamente ótimo.

Capítulo 7

Piece Tree

7.1 Introdução ao problema

Um dos tipos de software mais usados por um programador, talvez o mais usado, é o editor de texto. Um editor de texto deve ser capaz de lidar com um grande volume de inserções e deleções, tais operações podem acontecer em qualquer lugar do texto. É esperado que um editor de texto seja capaz de abrir diversos arquivos rapidamente e ocupando pouca memória.

Em 2018, o time de desenvolvedores do editor de texto *Visual Studio Code* (*VS Code*) decidiu mudar a estrutura de dados que era usada para armazenar o texto do arquivo. Tal mudança decorreu de reclamações do tempo gasto para abrir um arquivo e da falha do programa em abrir certos arquivos devido ao uso de memória do editor.

Neste capítulo, iremos explorar a estrutura de dados usado pelo *VS Code* anteriormente e como seus problemas foram resolvidos com uma nova estrutura de dados.

7.2 Vetores de Caracteres

Uma das estruturas mais básicas para ser usada em um editor de texto é um simples vetor de caracteres (*string*). Tal estrutura tem a grande vantagem de ser simples de ser implementada e, embora inserções/deleções no vetor tenham complexidade de tempo O(n) (n sendo o tamanho do vetor), o tempo amortizado é O(1). A operação de inserção/deleção em um ponto aleatório do texto tem complexidade O(n), porém não possui um bom tempo amortizado.

Dessa ideia simples surgem algumas outras para se armazenar texto. Uma delas, usada por um tempo no *VS Code*, é utilizar um vetor de *strings*, em que cada posição do vetor contém uma linha do arquivo. Tal modificação faz com que operações de edição tenham complexidade $O(l + m)$ (onde l representa o número de linhas no arquivo e m o tamanho da maior linha). Porém, como o time do *VS Code* descobriu, tal implementação pode usar muita memória se uma *string* na linguagem usada possuir uma quantidade mínima de memória grande e o arquivo sendo editado possuir muitas linhas.

7.3 Piece Table

Editores de texto modernos permitem que alterações recentes no texto possam ser rapidamente desfeitas, existem algumas maneiras de isso ser feito. Uma delas é usar a estrutura de dados conhecida como *piece table*.

Uma *piece table* é composta pelas seguintes partes: um vetor de caracteres que chamaremos de original - sendo que tal vetor irá conter o texto original do arquivo -, um vetor de caracteres chamado de adicional - que irá conter todos os caracteres escritos pelo usuário após o arquivo ter sido aberto - e um vetor de um tipo especial chamado de peças. Esse tipo especial, que chamaremos de nó, deve possuir 3 campos, um campo que irá armazenar uma posição, um campo que irá armazenar uma quantidade de caracteres e um campo para uma referência ao vetor original ou ao vetor adicional.

Suponha que desejemos abrir um arquivo que possui "Olá mun"como conteúdo, a *piece table* correspondente será da seguinte forma:

```
1   original: "Olá mun"
2   adicional: ""
3   peças: [(0, 7, original)]
```

se desejarmos adicionar os caracteres “do!” no final do arquivo, nossa estrutura final ficará da seguinte forma:

```
1   original: "Olá mun"
2   adicional: "do!"
3   peças: [(0, 7, original), (0, 3, adicional)]
```

como se pode notar, para fazer uma adição no final do arquivo, basta concatenar a frase desejada ao vetor adicional e inserir um novo elemento no vetor peças. Os campos do novo elemento inserido são iguais à posição em que a nova frase foi adicionada em adicional, o tamanho da nova frase e uma referência para adicional.

Agora suponha que desejamos inserir "meu querido "entre "Olá"e "mundo!", a estrutura se encontra da seguinte forma:

```
1   original: "Olá mun"
2   adicional: "do!meu querido "
3   peças: [(0, 4, original), (3, 12, adicional), (4, 3, original), (0, 3,
        adicional)]
```

como podemos observar, a nova parte é inserida no final do vetor adicional e uma peça é quebrada em duas, com uma nova peça inserida no meio.

Agora, para a deleção de partes do texto, suponha que um arquivo que contém o texto "Lorem ipsum dolor sit amet." acaba de ser aberto, teremos,

```
1   original: "Lorem ipsum dolor sit amet."
2   adicional: ""
3   peças: [(0, 27, original)]
```

se desejamos retirar “ ipsum dolor”, ao final da operação a estrutura se encontrará no seguinte estado:

```
original: "Lorem ipsum dolor sit amet."
adicional: ""
peças: [(0, 5, original), (17, 10, original)]
```

podemos observar que, só o vetor de peças foi alterado, a peça que correspondia ao que se desejava ser removido foi divida em duas partes. Uma apontando para o texto antes da parte removida e a outra parte correspondente ao texto após a parte removida.

7.4 Piece Tree

A primeira diferença entre a *piece tree* e a *piece table* é que, em vez de usar duas *strings*, um para guardar o conteúdo original do documento e outro para armazenar o conteúdo novo, são usadas múltiplas *strings* de 64 Kilobytes (KB) cada. Essa mudança é devido ao fato de que o *Vs Code* é feito usando *Node.Js* e quando a migração de estrutura de dados estava sendo feita, uma *string* não poderia ocupar mais de 256 MB de memória. Embora seja relativamente raro a necessidade de se abrir arquivos de texto tão grandes em um editor de texto, os desenvolvedores acharam válido dar suporte para arquivos de tal tamanho.

Outra diferença importante diz respeito a uma mudança feita no tipo Nó. Um Nó irá armazenar a posição relativa de todas as quebras de linha presentes na seção representada desse nó. Tal adição permite realizar a procura por uma linha específica mais rapidamente. Para maximizar o ganho de performance na procura por linhas, em vez de usar um simples vetor para armazenar os nós, é usada uma árvore de busca binária autobalanceada, em específico uma árvore rubro-negra.

De acordo com os testes realizados pelos desenvolvedores, a *piece tree* se saiu melhor que a implementação antiga em consumo de memória, tempo para abrir um arquivo e em edições em locais aleatórias e em sequência. Houve um caso em que a implementação antiga teve melhor performance, se o arquivo possui muitas edições, o tempo para extrair o conteúdo de uma linha do arquivo se provou maior. Os desenvolvedores decidiram por usar a *piece tree*, porém notaram que ainda havia mais otimizações a serem feitas.

7.5 Simulação

Para esta seção de simulação iremos usar *buffers* de no máximo 64 bytes. E nosso texto original será:

> Lorem ipsum dolor sit amet, consectetur adipiscing elit.
>
> Nunc mollis feugiat turpis id tempor.
>
> Maecenas sed scelerisque velit.

Tal texto inicial produz os seguintes dois *buffers*:

```
buffer1 = StringBuffer {
   value: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.\nNunc mo'
   lineBreaks: [57]
}
```

```
buffer2 = StringBuffer {
   value: 'llis feugiat turpis id tempor.\nMaecenas sed scelerisque velit.'
   lineBreaks: [31]
}
```

Com isso a *piece tree* terá forma igual a:

```
PieceTree {
   root: Raiz
   buffers: [buffer1, buffer2]
}
```

A árvore rubro-negra irá possuir dois nós, tais nós irão conter as seguintes peças:

```
Piece{
   indicieBuffer: 0
   começo: 0
   tamanho: 63
   númeroLinhas: 1
}
```

```
Piece{
   indicieBuffer: 2
   começo: 0
   tamanho: 63
   númeroLinhas: 1
}
```

Caso se deseje inserir o pedaço de texto *"aliquam as"*, deverá ser criado um novo *buffer*, já que o buffer2 já ocupa 64 bytes.

```
buffer3 = StringBuffer {
   value: ' aliquam as'
   lineBreaks: []
}
```

Esse novo *buffer* será adicionado na lista de *buffers*,

```
PieceTree {
   root: Raiz
   buffers: [buffer1, buffer2, buffer3]
}
```

Com essa inserção, um novo nó será inserido na árvore, tal nó irá possuir a peça:

```
Piece{
   indicieBuffer: 3
   começo: 0
   tamanho: 11
   númeroLinhas: 0
}
```

Tal nó será inserido na posição mais à direita da árvore, rebalanceando-a se necessário.

Após inserirmos a frase *"Locus domitarum, "* no ínicio do texto, o *buffer3* será a atualizado,

```
buffer3 = StringBuffer {
   value: ’ aliquam asLocus domitarum, ’
   lineBreaks: []
}
```

Como foi inserido o texto, iremos novamente inserir um novo nó na árvore. Como foi feito uma inserção no começo do texto, o novo nó será inserido na posição mais à esquerda da árvore. A peça correspondente a esse nó será igual a:

```
Piece{
   indicieBuffer: 3
   começo: 11
   tamanho: 16
   númeroLinhas: 0
}
```

Por fim, iremos mostrar a inserção da frase *“, botus clartis "* de tal forma que o texto resultante será da forma:

> Locus domitarum, Lorem ipsum dolor sit amet, consectetur adipiscing elit.
>
> Nunc mollis feugiat turpis id, botus clartis tempor.
>
> Maecenas sed scelerisque velit. aliquam as

Como ainda há espaço em *buffer3*, todo o texto será inserido nesse *buffer*,

```
buffer3 = StringBuffer {
   value: ’ aliquam asLocus domitarum, , botus clartis’
   lineBreaks: []
}
```

Em relação a árvore, o nó que possui a peça referente a parte do texto *’llis feugiat turpis id tempor.\nMaecenas sed scelerisque velit.’*,

```
Piece{
   indicieBuffer: 2
   começo: 0
   tamanho: 63
   númeroLinhas: 1
}
```

será dividido em dois,

```
Piece{
   indicieBuffer: 2
   começo: 0
   tamanho: 29
   númeroLinhas: 0
}
```

e

```
Piece{
   indicieBuffer: 2
   começo: 29
   tamanho: 35
```

```
5       númeroLinhas: 1
6   }
```

um novo nó com a peça que representa o novo pedaço de texto

```
1   Piece{
2       indicieBuffer: 3
3       começo: 28
4       tamanho: 15
5       númeroLinhas: 1
6   }
```

será inserido de tal forma que a árvore continue balanceada. Além disso, se os nós da árvore forem lidos do menor para o maior, o texto em seu estado atual será recuperado.

7.6 Conclusão

Neste capítulo, foram exploradas algumas estruturas de dados usadas por editores de texto para armazenar o conteúdo dos arquivos sendo editados. Em específico, foi discutido a utilização de um simples vetor de caracteres, o *gap buffer* e a *piece table*. Então, foi discutido como os desenvolvedores do editor de texto *VS Code* adaptaram a *piece table* para ela melhor se encaixar no *software* já existente, assim criando a *piece tree*.

Com este capítulo, espera-se que o leitor tenha sido familiarizado com algumas estruturas usadas por editores de texto, em específico o *gap buffer* e a *piece table*. Além de ter sido exemplificado como adaptar uma estrutura de dados para conseguir mais performance.

Capítulo 8

Completely Fair Scheduling

8.1 Introdução ao problema

Uma das muitas funções do Linux (e outros *kernels* de um sistema operacional) é o escalonamento (*scheduling*) de processos. *Scheduling* se refere ao ato de atribuir recursos para tarefas. Neste capítulo, iremos tratar da atribuição de núcleos para processos, será discutido um algoritmo simples de escalonamento e então será feito uma análise do *Completely Fair Scheduler*.

8.2 Terminologia

Antes de explorar algoritmos de escalonamento é preciso introduzir certa terminologia ao leitor. Recebe o nome de inanição ou *starvation* o evento em que um processo acaba por nunca ser executado pois um dado recurso não é disponibilizado para tal processo. Por exemplo, suponha que um grupo de usuários X possui prioridade maior para usar a impressora do que um outro grupo Y. Se sempre existir um processo do grupo X que necessita usar a impressora, processos do grupo Y que precisam usar esse mesmo recurso nunca irão ter acesso a ele, assim entrando em *starvation*.

Um processo se trata de uma instância de um programa em execução, ou seja um conjunto dos valores de variáveis e registradores. Já uma *thread* é um segmento de um processo, as diferenças de uma *thread* e de um processo são sutis, e, embora todas as *threads* de um programa compartilhem o mesmo espaço de memória, elas não compartilham registradores. Para o *scheduler* do Linux, não existe distinção entre um processo e uma *thread*, ambos são abstraídos para *tasks* (tarefas), assim não faremos a distinção entre eles quando se tratando de *schedulers*.

A definição de um algoritmo de escalonamento justo muda de acordo com o autor, entretanto podemos reduzir as definições para a seguinte forma: um algoritmo de escalonamento é chamado de justo se, recursos são atribuídos entre tarefas igualmente segundo alguma métrica.

8.3 Round Robin

Um dos algoritmos de escalonamento mais simples é o *round robin*. Tal algoritmo funciona da seguinte forma: os processos são inseridos em uma fila conforme são criados. Quando é hora de um processo ser executado, ele é retirado da fila e executado por uma quantia fixa de tempo (chamado de quanta). Se o processo ainda não estiver completado, ele é reinserido na fila (por fila entenda uma estrutura de dado *First in First Out*).

Além de fácil de ser implementado, o *round robin* não possui problemas de *starvation*, uma vez que todo processo é garantido de ser executado. Ele também é considerado um algoritmo justo. Porém, o algoritmo possui algumas desvantagens, não é possível dar maior prioridade para um dado processo. Calcular um quanta ótimo é um processo complicado, pois, se o quanta for muito pequeno, o sistema pode passar mais tempo fazendo mudanças de contexto do que executando programas; e se muito longo, o sistema pode se tornar não responsivo.

8.4 Completely Fair Scheduler

O *Completely Fair Scheduler* (CFS) (Escalonador Completamente Justo) é o *scheduler* padrão para processos do Linux desde 2008. Uma das características do *scheduler* anterior do Linux era o uso de heurísticas para identificar se um processo estava ativo ou inativo, tais cálculos se provaram complexos e propensos a erro, assim um novo *scheduler* sem heurísticas, porém igualmente veloz foi desenvolvido.

O CFS é considerado justo, pois os processos que passaram menos tempo sendo executados possuem maior prioridade para serem escalados, assim se tenta garantir que todos os processos sejam executados igualmente. É usada uma árvore rubro-negra para armazenar os processos (a chave da árvore é proporcional ao tempo que o processo passou sendo executado). Com isso, é possível adicionar e deletar processos em tempo O(log(n)) (n sendo o número de elementos na árvore). Também é mantida uma referência ao menor elemento da árvore, o que torna possível sua consulta e extração em tempo O(1).

Quando está na hora de um novo processo ser escolhido para ser executado, o processo com a menor chave é retirado da árvore. Tal processo será executado por x/n ms, onde n representa o número de processos na árvore e x é um intervalo de tempo pré-determinado. Quando o tempo especificado for esgotado, o processo irá parar de ser executado, o seu tempo de execução será atualizado e então o processo será inserido novamente na árvore.

Para evitar *starvation* de processos que estão rodando há muito tempo, é necessário iniciar novos processos com tempo de execução igual ao tempo mínimo presente na árvore. Caso os novos processos forem inseridos com tempo de execução igual a zero, é possível que processos antigos nunca sejam executados.

Ainda existem mais recursos no CFS, por exemplo, ele permite a existência de processos com maior ou menor prioridade. Para isso o tempo alocado para o processo ser executado pode ser modificado usando uma constante que indica o grau de prioridade. É também

possível expandir a ideia de justo para usuários/grupos, logo os processos de um usuário teriam a mesma quantia de tempo do que os processos de outro usuário.

8.5 Simulação

Para os exemplos de simulação do CFS, em vez de mostrar uma árvore rubro-negra, iremos utilizar um simples vetor ordenado para fins didáticos. Além disso, será usado 20 unidades de tempo como valor predeterminado para a alocação de tempo.

Suponha que em um dado momento a fila de prioridade do CFS se encontre da seguinte forma:

A	C	E	B
5	10	17	23

Figura 8.1: *Estado inicial da fila de prioridade*

Se um novo processo, F, for criado, ele será inserido na fila com chave igual a da menor chave presente na fila, neste caso a chave de F será igual a 5.

F	A	C	E	B
5	5	10	17	23

Figura 8.2: *Fila de prioridade após a inserção de um novo processo F*

Agora, suponha que o processo D, que estava sendo executado, é escalonado. D será reinserido na fila, com uma nova chave igual à sua chave anterior mais o tempo que D passou sendo executado e suponha que tal valor seja igual a 14.

F	A	C	D	E	B
5	5	10	14	17	23

Figura 8.3: *Estado da fila prioridade com a reinserção do processo D*

Com a CPU livre, um novo processo deve ser executado, tal processo será o que possui menor chave, o processo F (o processo A também poderia ter sido escolhido). Como existem 6 processos na fila, o processo F será executado por $\frac{20}{6}$ unidades de tempo, e a fila se encontrará da seguinte forma:

A	C	D	E	B
5	10	14	17	23

Figura 8.4: *Fila de prioridade após a remoção do processo de menor chave*

Suponha que F termine de executar, F não será reinserido na fila, e o processo A será escolhido para ser executado por $\frac{20}{5}$ unidades de tempo, resultando em uma fila de prioridade da seguinte forma:

C	D	E	B
10	14	17	23

Figura 8.5: *Fila de prioridade após a remoção do processo de menor chave*

8.6 Conclusão

Neste capítulo, foi explorado o algoritmo de escalonamento padrão do Linux, o *Completely Fair Scheduler*. Com este capítulo espera-se que o leitor tenha se familiarizado com noções de escalonamento de processo, além do funcionamento de um dos algoritmos de escalonamento mais utilizados do mundo.

Capítulo 9

Conclusão

O objetivo deste trabalho foi o estudo de estruturas de dados e algoritmos desconhecidos pelo autor (e espera-se que pelo leitor). No decorrer do texto, foram apresentados ao leitor diversos algoritmos e estruturas, como a *Piece table*, o *Bloom filter* e o algoritmo de procura de caminho usado pelo *Command & Conquer*. Foram fornecidos exemplos de como estruturas de dados foram alteradas para melhorar seus desempenhos, como foi o caso das mudanças feitas em uma *B+tree*.

Pela vasta quantia de *software* existente (muitos desses com código-fonte fechado), este trabalho é incompleto, pois existem muitos casos interessantes que mereciam ter sido abordados aqui, no entanto devido a uma quantia limitada de tempo (por limitada entenda não infinito) só uma pequena amostra pode ser explorada. Dito isso, essa pequena quantidade explorada abrangeu casos de uso desde de jogos eletrônicos até sistemas de gerenciamento de bancos de dados.

Referências

[Beyer e McCreight 1972] R Beyer e EM McCreight. "Organization and maintenance of large ordered indices". Em: *Acta Informatica* 1.3 (1972), pgs. 173–189 (citado na pg. 26).

[Bittman *et al.* 2018] Daniel Bittman *et al.* "Designing data structures to minimize bit flips on nvm". Em: *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE. 2018, pgs. 85–90.

[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 (citado na pg. 17).

[Botea *et al.* 2004] Adi Botea, Martin Müller e Jonathan Schaeffer. "Near optimal hierarchical path-finding." Em: *J. Game Dev.* 1.1 (2004), pgs. 1–30 (citado na pg. 45).

[Christensen *et al.* 2010] Ken Christensen, Allen Roginsky e Miguel Jimeno. "A new analysis of the false positive rate of a bloom filter". Em: *Information Processing Letters* 110.21 (2010), pgs. 944–949.

[Comer 1979] Douglas Comer. "Ubiquitous b-tree". Em: *ACM Computing Surveys (CSUR)* 11.2 (1979), pgs. 121–137 (citado na pg. 34).

[Knuth 1973] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. isbn: 0-201-03803-X (citado na pg. 34).

[Lomont 2003] Chris Lomont. "Fast inverse square root". Em: *Tech-315 nical Report* 32 (2003).

[Lyu 2018] Peng Lyu. *Text Buffer Reimplementation*. 2018. url: https://code.visualstudio.com/blogs/2018/03/23/text-buffer-reimplementation (acesso em 18/11/2021).

[Moroz *et al.* 2021] Leonid V Moroz, Volodymyr V Samotyy e Oleh Y Horyachyy. "Modified fast inverse square root and square root approximation algorithms: the method of switching magic constants". Em: *Computation* 9.2 (2021), pg. 21.

[Mullin 1983] James K Mullin. "A second look at bloom filters". Em: *Communications of the ACM* 26.8 (1983), pgs. 570–571 (citado na pg. 20).

[Overton 2001] Michael L Overton. *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.

[Pabla 2009] Chandandeep Singh Pabla. "Completely fair scheduler". Em: *Linux Journal* 2009.184 (2009), pg. 4.

[Robertson 2012] Matthew Robertson. *A brief history of invsqrt*. Department of Computer Science & Applied Statistics, 2012 (citado nas pgs. 6, 7).

[Rothenberg *et al.* 2010] Christian Esteve Rothenberg, Carlos A.B. Macapuna, Fabio L. Verdi e Maurício F Magalhães. "The deletable bloom filter: a new member of the bloom family". Em: *IEEE Communications Letters* 14.6 (2010), pgs. 557–559. doi: 10.1109/LCOMM.2010.06.100344 (citado na pg. 20).

[Sinha 2005] Prokash Sinha. "A memory-efficient doubly linked list". Em: *Linux Journal* 2005.129 (2005), pg. 10.

[Sommefeldt 2006a] Rys Sommefeldt. *Origin of Quake3's Fast InvSqrt() - Page 1*. 2006. url: https://www.beyond3d.com/content/articles/8/ (acesso em 21/03/2007) (citado na pg. 3).

[Sommefeldt 2006b] Rys Sommefeldt. *Origin of Quake3's Fast InvSqrt() - Part Two - Page 1*. 2006. url: https://www.beyond3d.com/content/articles/15 (acesso em 21/03/2007).

[Tanenbaum e Bos 2015] Andrew S Tanenbaum e Herbert Bos. *Modern operating systems*. Pearson, 2015.

[Tsou *et al.* 2012] Yao-Tung Tsou, Chun-Shien Lu e Sy-Yen Kuo. "Privacy-and integrity-preserving range query in wireless sensor networks". Em: *2012 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2012, pgs. 328–334.

[Walczyk *et al.* 2018] Cezary J. Walczyk, Leonid V. Moroz e Jan L. Cieslinski. "Improving the accuracy of the fast inverse square root algorithm". Em: *CoRR* abs/1802.06302 (2018). arXiv: 1802.06302. url: http://arxiv.org/abs/1802.06302.

[Wong *et al.* 2008] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari e Wey Fun. "Towards achieving fairness in the linux scheduler". Em: *Operating Systems Review* 42 (jul. de 2008), pgs. 34–43. doi: 10.1145/1400097.1400102.