Instituto Federal de Educação, Ciência e Tecnologia de Brasília, campus Taguatinga,
Campus Taguatinga

OTIMIZAÇÃO DE ÁRVORES RANGE-MIN-MAX PARA CONSULTAS SOBRE ÁRVORES SUCINTAS

Por

DANYELLE DA SILIVA OLIVEIRA ANGELO

Trabalho de Graduação

BRASÍLIA/2021

Danyelle da Siliva Oliveira Angelo

OTIMIZAÇÃO DE ÁRVORES RANGE-MIN-MAX PARA CONSULTAS SOBRE ÁRVORES SUCINTAS

Trabalho apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia de Brasília, campus Taguatinga, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Daniel Saad Nogueira Nunes

BRASÍLIA
2021

Danyelle da Siliva Oliveira Angelo
Otimização de árvores Range-min-max para consultas sobre árvores sucintas/ Danyelle da Siliva Oliveira Angelo. – BRASÍLIA, 2021-
Orientador Daniel Saad Nogueira Nunes

Trabalho de Graduação – Instituto Federal de Educação, Ciência e Tecnologia de Brasília, *campus* Taguatinga,, 2021.

1. Estrutura de dados sucintas. 2. Range min-max tree. I. Prof. Me. Daniel Saad Nogueira Nunes. II.Instituto Federal de Brasília. III. Otimização de árvores Range-min-max para consultas sobre árvores sucintas

CDU 004

Trabalho de Graduação apresentado por **Danyelle da Siliva Oliveira Angelo** ao curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia de Brasília, *campus* Taguatinga, sob o título **Otimização de árvores Range-min-max para consultas sobre árvores sucintas**, orientado pelo **Prof. Daniel Saad Nogueira Nunes** e aprovado pela banca examinadora formada pelos professores:

Prof. Me. Daniel Saad Nogueira Nunes
Departamento de Computação/IFB

Prof. Me. João Victor de Araujo Oliveira
Departamento de Computação/IFB

Prof. Dr. Felipe Alves da Louza
Departamento de Engenharia/UFU

BRASÍLIA
2021

I dedicate this thesis to all my family, friends and professors
who gave me the necessary support to get here.

Agradecimentos

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

When one finds a hard problem, the more complicated it is, the more one ought to work towards enlightening it's solution.

—MÁRCIO DE DEUS

Resumo

O desenvolvimento de novas aplicações de tempo real e o crescente aumento na produção de dados aliados à disparidade de desempenho entre processador e memória é um grande desafio para projetistas e desenvolvedores de software. Neste contexto, torna-se fundamental a utilização eficaz dos níveis superiores da hierarquia de memória, onde o tempo gasto para concluir uma solicitação do processador é menor e a capacidade de armazenamento é reduzida; essa utilização eficaz pode acontecer por intermédio das estruturas de dados sucintas, as mesmas possibilitam a representação e operação sobre um conjunto de dados de maneira eficiente, ao mesmo tempo em que possibilitam o seu gerenciamento em memórias mais rápidas e com capacidade de armazenamento menor.

A Range min-Max tree (rmM-tree) é um exemplo de sucesso dessas estruturas, construída na forma de uma árvore binária completa, a rmM-tree ocupa cerca de $n + O(\frac{n}{b}\log n)$ bits, e possibilita a realização de operações sobre os objetos aos quais representa em tempo $O(\log n)$ bits. Embora essa estrutura possua custo computacional teoricamente satisfatório, devido ao seu baixo fator de ramificação, podem ocorrer um grande número de transferências de dados entre cache e memória RAM. Nosso objetivo neste trabalho é minimizar o número de eventuais faltas de cache (*cache misses*) através da maximização da quantidade de dados enviados a cada transferência. Assim, esse trabalho se concentra no estudo da Range min-Max tree clássica e na otimização do uso da cache por essa estrutura mediante ao aumento do número de ramificações da mesma.

Palavras-chave: Estrutura de dados sucintas; Range min-Max tree; Árvores; Lacuna de desenvolvimento procesador-memória; Cache.

Abstract

The development of new real-time applications and the crescent increase data production combined with the disparity in performance between processor and memory is a major challenge for softwares designers and developers. In this context, it becomes essential to effectively use the higher levels of hierarchy of memory, where the time spent to complete a processor request is less and the storage capacity is reduced; the effective use of this hierarchy can take place through the succinct data structures, which allow the representation and operation on a set of data efficiently, while allowing the management of this data in faster memories and with less storage capacity.

The Range min-Max tree (rmM-tree) is an example of the success of these structures, built in the form of a complete binary tree, the rmM-tree occupies about $n + O(\frac{n}{b}\log n)$ bits, and makes it possible to perform operations on the objects it represents in time $O(\log n)$. Although this structure has theoretically satisfactory computational cost, due to its low branching factor, a large number of data transfers between cache and RAM can occur, our objective in this work is to minimize the number of possible cache misses, maximizing the amount of data sent with each transfer. Thus, this work focuses on the study of the classical Range min-Max tree interval and on optimizing the use of the cache by this structure, increasing its number of branches.

Keywords: Succinct data structures; Range min-Max tree; Trees; Processor-memory development gap; Cache.

Sumário

1

Introdução

A cada ano novas aplicações e dispositivos com formatos e recursos diversos são lançados no mercado, o crescimento de aplicações como as de transporte, dos serviços de streamings, a popularização de dispostivos IoT, o crescimento dos datacenters e do serviço em nuvem vêm contribuindo fortemente para o aumento na produção de dados (REINSEL D. E GANTZ, 2018).

Especializada em computação em nuvem, a DOMO realiza anualmente uma análise da quantidade de informação produzida na internet a cada *minuto*, abaixo vemos algumas das estatísticas relativas à esse estudo DOMO (2020):

- 41.666.667 mensagens são enviadas através do aplicativo de mensagens WhatsApp;
- 479.452 pessoas interagem com conteúdos da plataforma Reddit;
- 208.333 pessoas participam de conferências no serviço de chamada Zoom, ao passo que o Microsoft Teams conecta cerca de 52.083 usuários por minuto;
- 147.000 uploads de fotos são feitos no Facebook e 150.000 mensagens são enviadas na mesma plataforma;
- 28 faixas de música são incluídas no serviço de streaming Spotify a cada minuto;
- 500 horas de vídeos são enviadas ao youtube.

Prevê-se que essa quantidade de dados irá crescer nos próximos anos. De acordo com o relatório anual da CISCO (2020), em 2018 o número de usuários conectados à internet era de 3,9 bilhões, e até 2023 mais de 70% da população terá acesso à internet, atingido a marca de 5,3 bilhões de usuários. A International Data Corporation (IDC) prevê que em 2025, 75% (6 bilhões) da população mundial interaja com essas aplicações todos os dias, fazendo com que a quantidade de dados produzidos cresça de 33 Zetabytes (ZBs) em 2018, para 175 ZBs em 2025 (REINSEL D. E GANTZ, 2018).

Esee crescimento da produção de dados tem relação direta com o surgimento de novos dispositivos. O grande problema é que existe uma lacuna de desempenho entre CPU e memória. Essa lacuna se deve principalmente ao fato de que os fabricantes de processadores estão focados em obter uma lógica rápida, que acelera a comunicação, ao passo que os fabricantes de memória

objetivam uma capacidade de armazenamento de dados maior (EFNUSHEVA D.; CHOLAKOSKA, 2017). Essa disparidade é um grande desafio para projetitas e desenvolvedores, pois como afirmam EFNUSHEVA D.; CHOLAKOSKA (2017), esta faz com que seja criado um atraso na comunicação entre CPU e memória (gargalo de Von-Neumann), sobretudo quando se trabalha com um grande conjunto de dados. Diversas soluções foram e estão sendo desenvolvidas para contornar o problema dessa lacuna, entre elas o uso de computação paralela em nível de instrução, hierarquia de memória multinível, *smarter memories*, técnicas específicas de tolerância à latência, e compressão de dados. (MAHAPATRA N. R. E VENKATRAO, 1999; EFNUSHEVA D.; CHOLAKOSKA, 2017)

Como mostra NAVARRO (2016), estruturas de dados sucintas são uma forma de compressão de dados, estas representam a informação de maneira eficiente, usando um espaço próximo ao limite inferior estabelecido pela Teoria da Informação, possibilitando ainda operações sobre seus objetos de modo que não seja necessário a descompactação dos mesmos, o que a torna mais eficiente do que os algorotimos de compatação clássicos que precisam de armazenamento e tempo extra para descompactar os objetos sobre os quais operam. Esses fatores fazem com que as estruturas de dados sucintas sejam amplamente usadas em sistemas como os de mecanismo de busca, ou sistemas que trabalham com dados geográficos. Estruturas de dados sucintas também são essenciais para lidar com dispositivos onde a quantidade de memória é limitada, como dispositivos IoTs e outros embarcados (NAVARRO, 2016). Por fim, mesmo que seja necessário reter parte do conjunto de dados em um nível mais baixo da hierarquia de memória, onde o tempo para obter uma informação é maior, o uso de estruturas de dados sucintas ainda possibilitará aplicações eficientes, haja vista que o seu uso minimizará o número de acessos a memórias mais lentas.

Trabalharemos especificamente com o estudo de árvores sucintas, uma das estruturas mais populares no campo da Ciência da Computação, tanto no caso das representações clássicas quanto para as sucintas. De acordo com ARROYUELO (2010), as implementações de árvores sucintas existentes na literatura podem diferir em sua funcionalidade, variando daquelas que suportam navegação de um nó filho para um nó pai ou aquelas que suportam operações como obter o menor ancestral comum de dois nós, até àquelas que suportam um conjunto completo de operações, podendo variar em relação ao espaço ocupado, indo de $O(n/(\log\log n)^2)$ à $O(n/polylog(n))$ bits.

A nossa contribuição consiste na otimização de uma estrutura de dados sucinta já existente: a Range min-Max tree (rmM-tree), proposta por SADAKANE K. E NAVARRO (2010). Essa estrutura fornece suporte à diversas operações sobre árvores ordinais compactas. Em sua versão estática, ela pode ser construída usando apenas $n + O(\frac{n}{b}\log n)$ bits de espaço, sendo capaz de realizar operações em tempo $O(\log n)$(SADAKANE K. E NAVARRO, 2010). Mas mesmo ao usarmos estruturas de dados sucintas tão eficiente quanto a rmM-tree, quando trabalhamos com grandes volumes de dados, parte dos mesmos podem ser armazenados nos níveis mais inferiores da hierarquia, sobretudo quando se trabalha em um nível com capacidade

de armazenamento tão pequeno quanto a cache. No caso da rmM-tree, temos como agravante o fato de que a mesma é uma implementação sobre árvores binárias, que por sua vez possuem fator de ramificação pequenos, o que pode gerar um alto número de transferências entre cache e RAM quando uma informação não é encontrada de imediato.

Com base no exposto, e estudos realziados, buscaremos maximizar a quantidade de informações relevantes nos nós da rmM-tree clássica, bem como o número de ramificações dessa estrutura, visando minimizar o número de eventuais trocas de dados entre cache e memória RAM. Para tanto propomos uma versão da Range min-Max tree k-ária.

1.1 Objetivos

O objetivo central deste trabalho é propor uma versão alternativa da Range min-Max tree, visando diminuir a quantidade de transferência de dados entre níveis da memória, melhorando assim o desempenho geral da estrutura.

Através dessa proposta temos como objetivos secundários: estudar e compreender diferentes representações sucintas de árvores, seus fundamentos e benefícios; compreender os impactos da hierarquia de memória e os benefícios da utilização adequada dos recursos fornecidos por esta arquitetura. Estudaremos a fundo a estrutura proposta por SADAKANE K. E NAVARRO, visando compreendendo melhor as operações de navegação e consulta em árvores gerais fornecidas por esta. Implementaremos então a versão original da rmM-tree buscando abranger todas as operações suportadas na literatura, e também a versão alternativa da rmM-tree, usando uma estrutura k-ária, afim de verificar o objetivo central do trabalho, comparando deste modo o desempenho de ambas as estruturas na hierarquia de memória.

Nesse sentido, faz parte também dos objetivos secundários, compreender e aprender a utilizar corretamente bibliotecas e frameworks que ajudarão a embasar a análise experimental deste trabalho, tais como a biblioteca *Succint Data Structure Library* (SDSL), e os frameworks *Google tests* e *Google benchmark*.

De modo a atingir os objetivos citados, o capítulo traz um estudo sobre hierarquia de memória, estrutura de dados sucintas e as características e operações suportadas pela range min-Max tree, o capítulo 3 apresenta a rmM-tree k-ária, expondo suas principais características e diferenças em relação à estrutura clássica. O capítulo 4, discorre a respeito dos testes experimentais, metodologia usada para os mesmos e resultados alcançados, por fim o capítulo 5 expões as nossas conclusões em relação aos objetivos e os resultados, bem como as nossas perspectivas em relação ao que foi desenvolido até o momento de escrita deste trabalho.

2

Referencial teórico

Quando trabalhamos com um conjunto de dados relativamente grande, parte deste tende a ser distribuído entre os níveis da hierarquia de memória, podendo acarretar em um alto número de transferência de dados entre níveis e consequentemente numa piora do desempenho geral do sistema.

Este capítulo discorre sobre a importância do estudo e implementação de estruturas de dados sucintas afim de sanar o problema do alto número de transferências de dados entre diferentes níveis da memória, apresentando também conceitos básicos de representações que são amplamente usadas na construção dessas estruturas.

2.1 Avanços tecnológicos e estrutura de dados sucintas

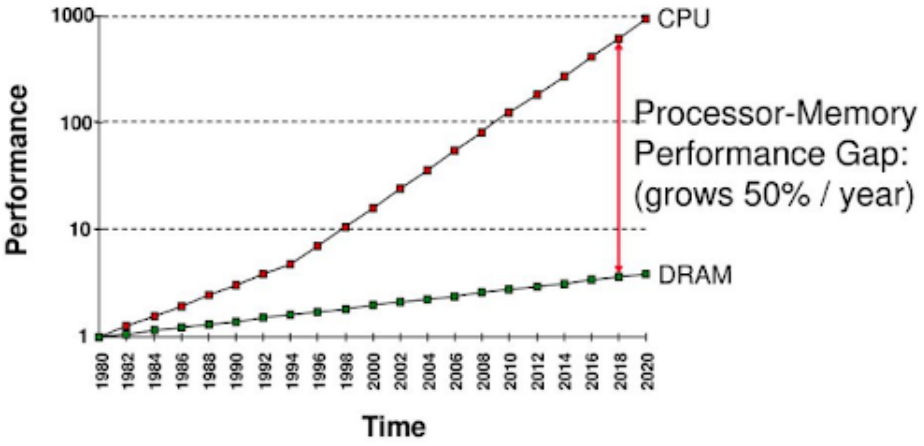
A divisão da indústria de microprocessadores e memória gera o que conhecemos como gargalo de comunicação entre processador e memória principal - gargalo de Von-Neumann - isso se deve em partes ao fato de que a indústria de processadores visa desenvolver uma lógica que acelera a comunicação, ao passo que os fabricantes de memória tem por objetivo aumentar a capacidade de armazenamento de dados (EFNUSHEVA D.; CHOLAKOSKA, 2017), esses diferentes objetivos fazem com que o desempenho desses dois componentes cresça com uma diferença de 50% ao ano, como ilustrado na Figura 2.1.

Apesar das melhorias feitas por cada uma destas indústrias ao longo dos anos, a disparidade de desempenho dos dois componentes fazem com que o processador imponha demandas significativas à memória, que em um cenário real não podem ser supridas, gerando assim um sistema desequilibrado, e então, devido a baixa largura de banda da memória em comparação ao alto desempenho do processador temos um aumento no tempo gasto para concluir uma solicitação, afetando diretamente o desempenho das aplicações. Para suprir esse desequilíbrio os projetistas de computadores criaram o que conhecemos como hierarquia de memória, que consiste em conectar o processador

> *"a um conjunto hierárquico de memórias, cada uma das quais maior, mais lenta e mais barata (por byte) do que as memórias mais próximas ao processador"(CARVALHO, 2002, tradução nossa).*

—‘

Figura 2.1: Lacuna de desempenho entre processador e memória a cada dois anos, começando de 1980



Retirado de: http://mirkwood.cs.edinboro.edu/ bennett/class/csci312/fall2018/notes/five/one.html

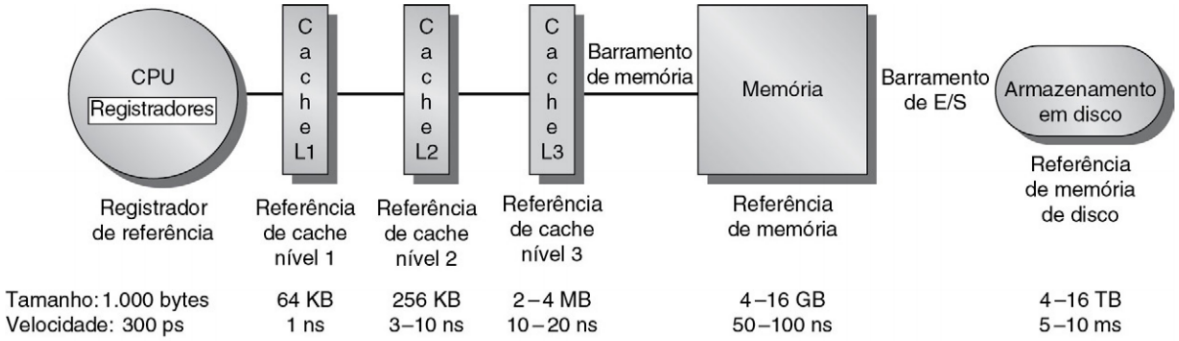
Nas arquiteturas computacionais modernas essa hierarquia é formada por: registradores, cache, memória primária e memória secundária. A Figura 2.2 demonstra uma hierarquia de memória e os seus respectivos valores de latência e espaço para cada nível de um servidor e um dispositivo móvel hipotético. Pela figura é possível observar que, à medida que nos distanciamos do processador, as unidades de tempo sobem à fatores de 10^9 — de picossegundos para milissegundos — e as unidades de tamanho mudam por fatores de 10^{12} — de bytes para terabytes.

Devido a baixa latência que as memórias mais distantes do processador possuem, torna-se ideal para obter um desempenho efetivo das operações sobre um conjunto de dados, trabalhar nos níveis superiores da hierarquia de memória (NAVARRO, 2016). A grande problemática é que como vemos em CARVALHO (2002), as memórias com latência menor possuem capacidade de armazenamento também menor, tornando praticamente inviável manipular grandes conjuntos de dados nestes níveis, o que é cada vez mais comum nos dias atuais. Uma possível solução para tanto é operar sobre os dados em sua forma compactada e conforme cita também COIRA, a melhor maneira de fazer isso é através do uso das estruturas de dados sucintas.

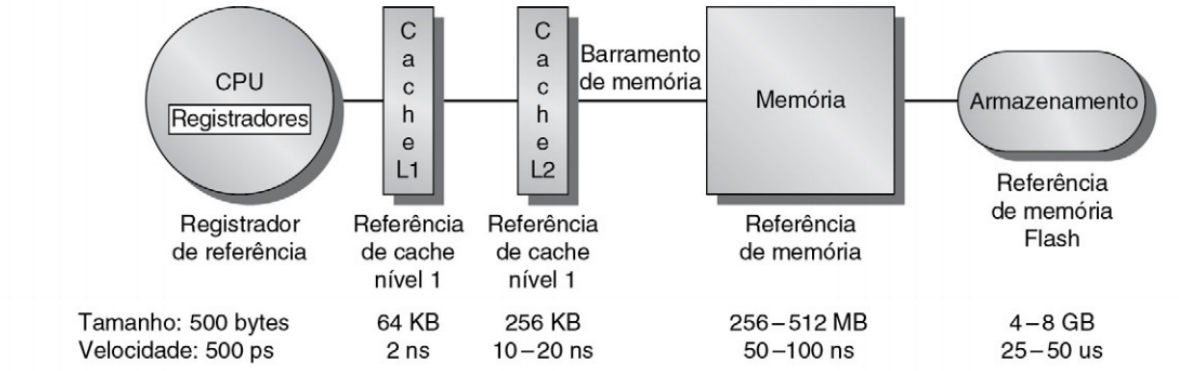
Possuindo relação direta com a Teoria da Informação[1], uma estrutura de dados é dita sucinta se ela pode representar informações usando um espaço próximo à entropia[2] determinada por essa teoria, que como mostra (NAVARRO, 2016), em seu livro, no pior caso é de $\log_2 |U|$ bits para um objeto de cardinalidade U. Além disso, essa estrutura deve fornecer suporte a uma série de operações primitivas sobre os seus objetos de modo eficiente. Por fim, uma estrutura de dados sucinta é considerada mais eficiente do que outros algoritmos de compactação clássicos, porque estes precisam descompactar os dados antes de operar sobre os mesmos, tornando inviável a

[1] Teoria matemática proposta inicialmente por Claude Shannon que busca quantificar a informação.
[2] Número mínimo de bits necessários para diferenciar um objeto em um conjunto de dados.

Figura 2.2: Os níveis em uma hierarquia de memória em um servidor (a) e em um dispositivo pessoal móvel (PMD) (b).



(a) Hierarquia de memória de um servidor



(b) Hierarquia de memória de um dispositivo móvel

Fonte: HENNESSY J. L. E PATTERSON (2014)

manipulação de grandes conjuntos de dados em memórias como a cache e a RAM.

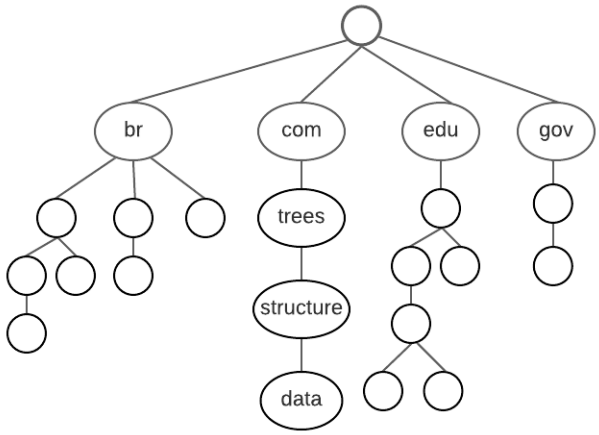
2.2 Árvores Sucintas

As árvores são uma das estruturas de maior sucesso na ciência da computação. Isso se deve ao fato de que essa estrutura é de simples compreensão e possibilita operações de busca, inserção e remoção de forma eficiente. Uma árvore $T = (V,E)$ é uma estrutura de dados hierárquica, ou seja não linear, formada por vértices (V) e arestas (E). Assim, árvores são definidas também como grafos, sendo eles conexos, não dirigidos e acíclicos, ou seja para qualquer dois vértices em T existe um único e simples caminho. (CORMEN, 2012).

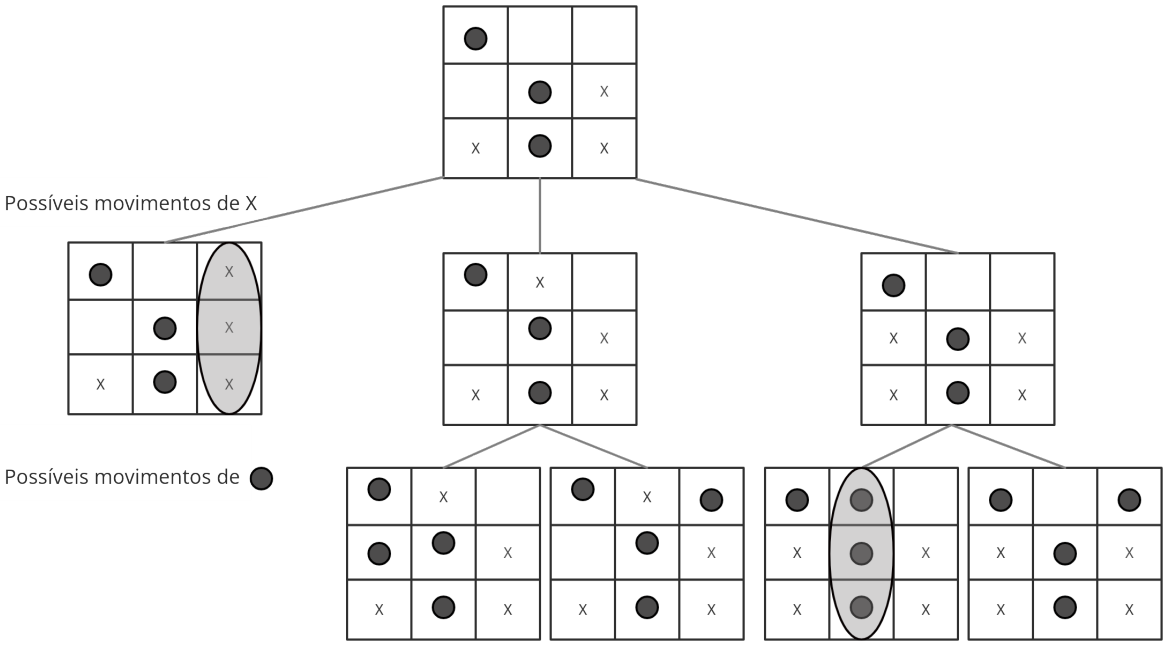
Encontramos diversas aplicações construídas a partir destas estruturas, entre elas aplicações de banco de dados e tradutores de idiomas (CORMEN, 2012). Árvores também são amplamente usadas no campo de aprendizagem de máquina: são as chamadas árvores de decisão, que auxiliam desde a escolha do melhor movimento em um jogo de xadrez à operações em sites de comércio eletrônico (RUSSELL S. E NORVIG, 2013). A Figura 2.3 exemplifica a forma gráfica de uma árvore que armazena a estrutura do domínio *data.structure.trees.com* (DNS são organizados em árvores, tendo o seu sufixo armazenado na raíz) e parte da representação de uma

árvore de decisão de um jogo da velha.

Figura 2.3: Exemplos de representação gráficas de árvores



(a) Árvore de DNS, a subárvore enraízada no nó **com** define o domínio *data.structure.trees.com*



(b) Segmento de uma árvore de decisão para um jogo da velha

Mas mesmo estruturas eficientes como as árvores podem se tornar inviáveis para determinadas aplicações. A forma como estas são construídas afim de operar sobre os objetos podem muitas vezes ocupar um espaço ainda maior do que os dados ocupariam originalmente. Para reconhecer este problema NAVARRO (2016) traz como exemplo uma comparação do espaço ocupado pelo genoma humano armazenado sem estruturas adicionais e o espaço ocupado pelo mesmo usando uma árvore de sufixos para viabilizar operações de montagem de genomas. Com 3,3 bilhões de nucleotídeos, se usarmos 2 bits para armazenar cada base nitrogenada necessi-

taremos de aproximadamente 825 megabytes, assim podemos reter o DNA por completo em qualquer memória principal, no entanto, usando a árvore de sufixos, cada nucleotídeo precisará de 10 bytes para ser representado, o que nos leva a uma estrutura que ocupa um espaço igual à 33 gigabytes, o que torna inviável o processamento do genoma em memória principal na maioria dos computadores comuns.

É nesse ponto que entram as estruturas de dados sucintas. Como já vimos, estas são capazes de armazenar tanto as informações, como as estruturas de dados que atuam sobre elas usando um espaço reduzido. No caso das árvores que é o objeto do nosso estudo, a sua representação clássica com ponteiros ocupa $O(n \log n)$ bits[3], enquanto existem representações sucintas, abordadas nas Seções 2.3 e 2.4.1, que ocupam cerca de $2n + o(n)$ bits.

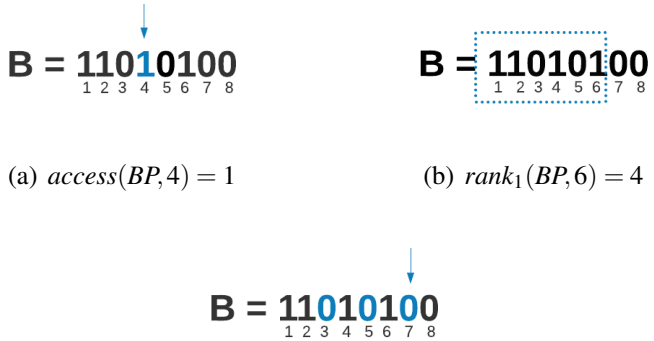
2.3 Vetores de Bits

Um vetor de bits $BP[0, n-1]$ é uma sequência sobre o alfabeto $\Sigma = \{0, 1\}$. É interessante que as seguintes operações sejam executadas sobre os vetores de bits (NAVARRO, 2016):

- $access(BP, i)$: retorna o i-ésimo bit do vetor BP, com $0 \leq i < n$;
- $rank_v(BP, i)$: seja $v \in \{0, 1\}$, e $0 \leq i < n$, esta operação retorna o número de ocorrências de v no intervalo $BP[1, i]$.
  Sendo que a seguinte relação de equivalência é válida: $rank_0(BP, i) = i - rank_1(BP, i)$.
  Um caso particular dessa operação é $rank_v(BP, 0) = 0$;
- $select_v(BP, i)$: dado $v \in \{0, 1\}$ e $0 \leq i < n$. *select* retorna a posição do i-ésimo bit v em B, sendo que assim como em *rank* por definição $select_v(BP, 0) = 0$.

A Figura 2.4 traz exemplos das operações listadas.

Figura 2.4: Operações de *rank*, *select* e *access* sobre $B = 11010100$



(a) $access(BP, 4) = 1$

(b) $rank_1(BP, 6) = 4$

(c) $select_0(BP, 3) = 7$

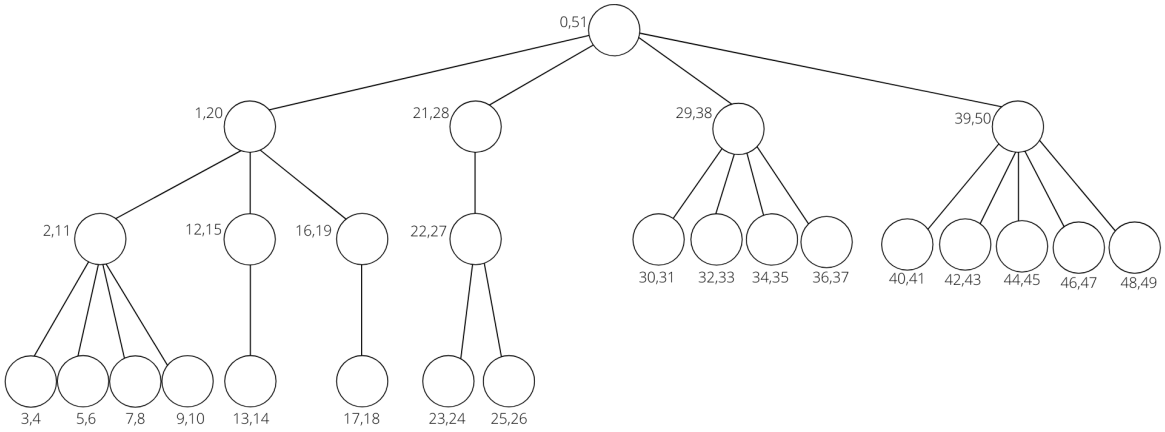
[3]Neste trabalho, quando não indicado, estaremos trabalhando com o logarítmo na base 2.

2.4 Representação de árvores sucintas

Existem diversas formas de representar uma árvore de maneira sucinta, abaixo listamos algumas destas, sendo que o nosso foco neste trabalho será a primeira forma, representação sucinta via parênteses balanceados.

2.4.1 Parênteses Balanceado (BP)

Figura 2.5: Representação de uma árvore geral T usando parênteses balanceados: fazendo um percurso pré-ordem em T escrevemos um parênteses de abertura quando um nó é visitado pela primeira vez, e um de fechamento no percurso de volta após atravessar sua subárvore (MUNRO J. I. E RAMAN, 1997)



(((( ) ( ) ( ) ( )) (( )) (( ))) ((( ) ( ))) (( ) ( ) ( ) ( )) (( ) ( ) ( ) ( ) ( )))
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Uma sequência de parênteses balanceados (BP) consiste em uma string de tamanho igual à $2n$, sendo n parênteses de abertura '(', e n parênteses de fechamento ')'. Essa estrutura descreve uma relação de hierarquia/contenção, e portanto é amplamente utilizada para a representação de árvores.

Para representar um árvore ordinal T usando essa estrutura, tomamos um vetor de bits de tamanho igual à $2n$ ($BP[0, 2n-1]$), onde n é o número de nós da árvore. Realizamos então um percurso sobre T em pré-ordem, e sempre que um nó for alcançado pela primeira vez, um parênteses de abertura (representado pelo bit 1) é inserido em BP, ao término da exploração das subárvores deste nó, um parênteses de fechamento (bit 0) é inserido em BP (ARROYUELO, 2010). Na figura Figura 2.5 vemos uma árvore com 26 nós e sua representação equivalente em parênteses balanceados.

Usando a representação de parênteses balanceados, podemos fornecer suporte às operações a seguir para diversas estruturas de dados sucintas:

- $findclose(BP,i)$: retorna a posição j do parênteses de fechamento correspondente ao i-ésimo parênteses de abertura;

- $findopen(BP,i)$: retorna a posição j do parênteses de abertura correspondente ao i-ésimo parênteses de fechamento;

- $excess(BP,i)$: retorna a "diferença entre o número de parênteses abertos e fechados até a posição i". (MUNRO J. I. E RAMAN, 1997, tradução nossa)

MUNRO J. I. E RAMAN (1997) proporam uma solução em árvores binárias usando a representação de parênteses balanceados, onde além das operações já citadas, oferece também suporte a operação de $enclose(i)$, que retorna o pai do nó i, e a operação $double\_enclose(i,j)$ (que equivlae à operação lca em árvores) que por sua vez retorna o pai dos nós i e j.

Através da operação *rank* podemos viabilizar a operação de *excess* facilmente, como mostrado abaixo.

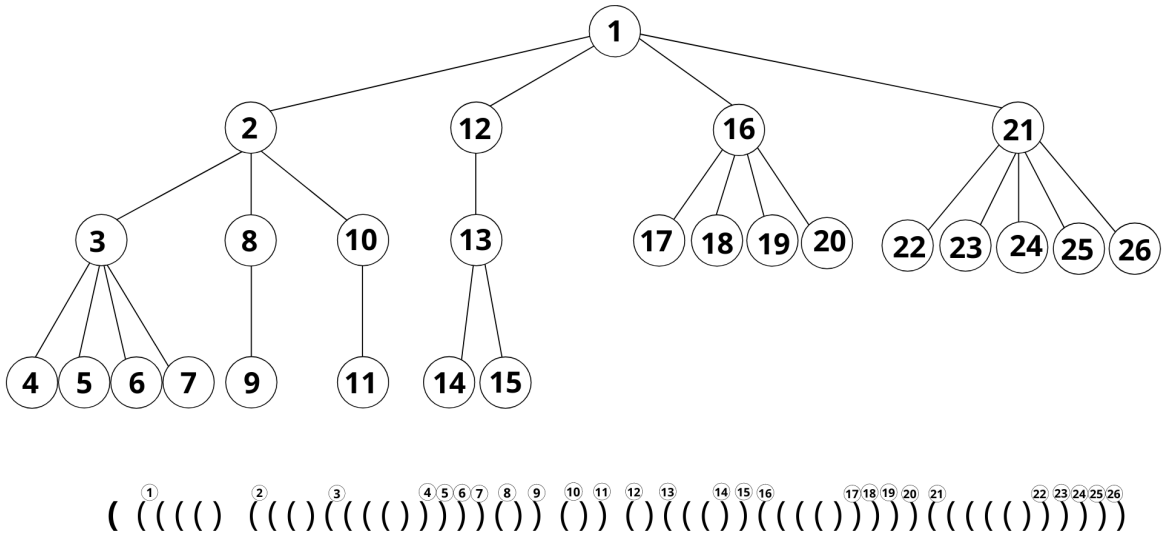
$$\begin{aligned} excess(BP,i) &= rank_1(BP,i) - rank_0(BP,i) \\ &= rank_1(BP,i) - (i - rank_1(BP,i)) \\ &= 2 \cdot rank_1(BP,i) - i \end{aligned}$$

Por fim, através das operações definidas em vetores de bits e parênteses balanceados podemos fornecer suporte às diversas operações sobre árvores, como a obtenção do tamanho da subárvore de um nó i, que pode ser feita através de: $(findclose(BP,i) + i - 1)/2$. Esta expressão retornará a quantidade de nós codificados dentro do intervalo de contenção de i. Podemos ainda averiguar se um nó j da árvore é ou não um nó folha, bastando verificar se o elemento que segue imediatamente à j é um parênteses de fechamento, isso pode ser feito através da operação de $access(BP, j+1)$, que irá retorna o k-ésimo bit de BP, com $k = j+1$.

2.4.2 Unary Degree Sequence (DFDUS)

Outra forma de representar árvores é atráves da *Sequência de grau unário*, que também faz o uso de uma travessia pré-ordem (como BP) para construir a árvore. A diferença é que a para codificar um nó, inserimos i parênteses de abertura (onde i é o número de filhos deste nó) e 1 parênteses de fehcamento. Desse modo, conforme afirma ARROYUELO, "o nó passa a ser representado pela posição de seus i parênteses". O autor mostra que a sequência resultante ainda é balanceada, se adicionarmos um parênteses de abertura ao ínicio da mesma. A Figura 2.6, mostra a representação equivalente à de parênteses balanceadas, com *DFDUS* para a árvore geral mostrada acima.

Figura 2.6: Representação de uma árvore geral T usando DFDUS



2.4.3 Level-ordered Unary Degree Sequence (LOUDS)

Nesta representação, pecorremos os nós em largura, adicionando assim como em *DFDUS* *i* parênteses de abertura ao visitarmos um nó pela primeira vez, e 1 parênteses de fechamento. Novamente com *i* represetando o número de filhos daquele nó.

2.5 Range min-Max Tree

A Range min-Max-tree (rmM-tree) é construída na forma de uma árvore binária completa (o que nos permite usar uma heap[4] sem ponteiros), onde cada um de seus nós armazenam valores de excesso calculados dentro de um intervalo de um vetor da representação de parênteses balanceados. Esse vetor deve representar a relação de contenção dos nós da árovre ordinal sobre a qual devemos operar, para tanto podemos usar a estrutura de vetores de bits ou a de parênteses balanceados que expomos anteriormente na Figura 2.5.

A construção da rmM-tree segue uma abordagem *bottom-up*, onde a cada nó, calcula-se primeiro o intervalo a ser coberto, para então definir os valores de excesso. Após construir os nós folhas dessas estrutura, repetimos o processo descrito para os nós internos (e raíz), sendo que o intervalo coberto por cada nó é a união de área de seus nós filhos.

Como cada nó folha está associado a um único intervalo, o tamanho dessa estrutura está também relacionada a quantidade e ao tamanho dos intervalos cobertos pela mesma. Esses intervalos são obtidos da seguinte maneira, dado um vetor de entrada *BP*, de tamanho igual à n, sendo *BP* usado para a codificação de uma árvore ordinal T, é definido um tamanho de intervalo b (chamado também de tamanho de bloco), e então divide-se n por b, obtendo assim a

[4] "Objeto arranjo que pode ser visto como uma árvore binária quase, onde cada nó da árvore corresponde a um elemento do arranjo."CORMEN (2012)

a quantidade de folhas da rmM-tree. Considerando que a altura de uma árvore binária é dada por $\lceil \log n \rceil$ e que o vetor de entrada B ocupa n bits, temos assim que a rmM-tree tem uma ocupa um espaço próximo à $n + O(\frac{n}{b} \log n)$ bits, fazendo $b = \log^2 n$, temos uma complexidade de espaço igual à $n + O(n/\log n)$. Por fim, como mostra NAVARRO (2016) em seu livro, na prática todas as operações sobre a rmM-tree podem ser realizadas em tempo $O(\log n)$ - ou ainda em tempo $O(\log \log n)$ em uma versão mais refinida da estrutura - a lista completa dessas operações é apresentada na Tabela 2.1 deste capítulo.

2.5.1 Registros

Os valores de excesso definido em cada nó da rmM-tree são essenciais para a realização de operações de consulta de maneira eficiente, são neles em que essa estrutura se baseia. Em seu livro, NAVARRO , define 4 valores de excesso que viabilizam essas operações. Mostraremos as definições de cada um destes a seguir. Suponha que um nó v cubra um intervalo $[s,e]$ do vetor de entrada BP de parênteses balanceados, define-se então:

- *R[v].e*: excesso total no intervalo $[s,e]$

  $R[v].e = excess(e) - excess(s-1)$.

- *R[v].m*: excesso mínimo local

  $R[v].m = \min\{excess(i) - excess(s-1) | s \leq i \leq e\}$.

- *R[v].M*: excesso máximo local

  $R[v].M = \max\{excess(i) - excess(s-1) | s \leq i \leq e\}$.

- *R[v].n*: é definido pelo número de vezes que o excesso mínimo ocorre dentro do intervalo coberto. Assim, suponha m o vetor de excessos sobre o intervalo $[s,e]$, então: $R[v].n = |\{BP[i] = R[v].m | s \leq i \leq e\}|$

Agora que temos a definição de cada campo da Range min-Max tree (estas definem os valores dos nós folhas), podemos analisar a construção de seus nós internos. Como já dito, estes nós são construídos a partir da união das áreas cobertas por seus nós filhos, e como nossa estrutura é construída através de um vetor, sabemos que os filhos de um nó v é dado pelos nós $2v+1$ e $2v+2$, isso nos leva as seguintes recorrências:

- $R[v].e = R[2v+1].e + R[2v+2].e$
- $R[v].m = min(R[2v+1].m, R[2v+1].e + R[2v+2].m)$
- $R[v].M = max(R[2v+1].M, R[2v+1].e + R[2v+2].M)$
- $R[v].n = \begin{cases} R[2v+1].n, & \text{se } R[2v+1].m < R[2v+1].e + R[2v+2].m; \\ R[2v+2].n, & \text{se } R[2v+1].m > R[2v+1].e + R[2v+2].m; \\ R[2v+1].n + R[2v+2].n, & \text{se } R[2v+1].m = R[2v+1].e + R[2v+2].m. \end{cases}$

Examplo 1: Para melhor elucidar a construção dessa estrutura usaremos o exemplo da Seção 2.4.1 e demonstraremos o cálculo dos campos e dos intervalo de um nó folha, mostraremos também como é construído os campos de um dos nós internos da rmM-tree. No exemplo em questão, temos um vetor de entrada com 52 parênteses balanceados, definiremos um tamanho de bloco $b = 4$, o que implica que a árvore terá:

- $r = n/b \rightarrow r = 52/4 = 13$ folhas;
- 12 nós internos, pois $r - 1 = 12$;
- Altura $h = \lceil \log r \rceil \rightarrow h = \lceil \log 13 \rceil = 4$;
- Como o número de folhas não é uma potência de 2, temos que as folhas 0 até $2 * r - 2^h - 1 = 9$ estão agrupadas no último nível da árvore, e as outras $2^h - r =$ folgas estão agrupadas à direita no nível anterior.

Assim temos os seguintes valores para a folha 0 (nó 15) da árvore:

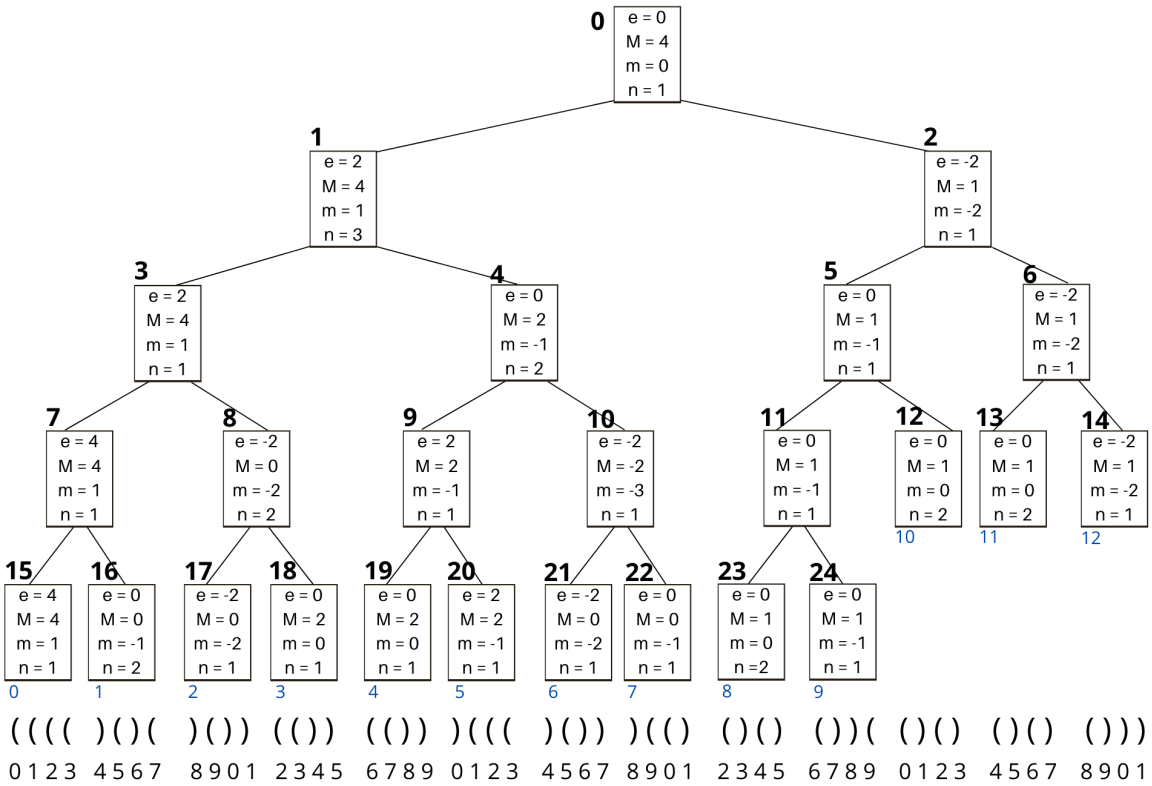
- Área de cobertura: $[(i-1) \cdot b + 1, i \cdot b] = [1,4]$
- Excesso global: $R[15].e = excess(4) = 2 \cdot rank_1(BP,4) - 4 = 4$
- Excesso mínimo: $R[15].m = min(1,2,3,4) = 1$
- Excesso máximo: $R[7].M = max(1,2,3,4) = 4$
- Número de vezes que o excesso mínimo ocorre no intervalo: $R[7].n = count((1,2,3,4),1) = 1$

Mostraremos agora o cálculo do 7º nó interno do nosso exemplo, este cobre as folhas 0 e 1 da árvore (nós 15 e 16, respectivamente), com base nas definições anteriores temos então:

- Área de cobertura: $BP[1,4] \cup BP[5,8]$
- Excesso global: $R[7] = R[15].e + R[16].e = 4 + 0 = 4$
- Excesso mínimo: $R[7].m = min(R[15].m, R[15].e + R[16].m) = (1, 4-1) = 1$
- Excesso máximo: $R[7].M = max(R[15].M, R[15].e + R[16].M) = (4, 4+0) = 4$
- Número de vezes que o excesso mínimo ocorre no intervalo: $R[7].n = R[15].n = 1$ pois $R[15].m < R[15].e + R[16].m$

O processo mostrado acima deve ser repetido para os demais nós da árvore até que cheguemos a raíz, dando origem a árvore mostrada na Figura 2.7.

Figura 2.7: rmM-tree clássica,com tamanho de bloco igual à 4. A estrutura foi construída a partir dos 52 parênteses balanceados mostrados na parte inferior



O processo de construção da rmM-tree é mostrado com mais detalhes no algoritmo 1. No algoritmo apresentado, usamos uma tabela C, o uso dessa tabela não é obrigatório, mas ela otimiza o tempo gasto para construir e operar sobre a rmM-tree, como mostra NAVARRO. A construção desta, se dá do seguinte modo, define-se primeiro uma constante w, tal que $b \bmod w = 0$, e cria-se então uma tabela $C[0, 2 * w]$, onde cada entrada da tabela armazena valores excesso definidos pela estrutura da rmM-tree, cada elemento $C[x]$ é calculado com base nos w bits que formam x.

Examplo 2: Suponha $w = 4$ e $x = 3$, temos assim que:

$$3_2 = 0011,$$

logo:

- $C[x].e = 0; C[x].M = 0; C[x].m = -2; C[x].n = 1$

Fazer a pré-computação dessa tabela elimina a necessidade computar iterativamente os valores de excesso mínimo e máximo, a cada inspeção de bits. Assim, no momento da construção dos nós folhas, ou durante a inspeção dos bits que a compõe, basta ler os w bits que

queremos inspecionar, convertêr-los para a base 2, e então realizar uma consulta na tabela C, que conforme mostra NAVARRO (2016) leva tempo $O(\frac{n}{\log n})$. A função *bitsread*, atua junto à tabela C otimizando o tempo de resposta das nossas operações, e por esse motivo é usada durante todo o nosso projeto, ela recebe como parâmetro um índice i, e lê i e os $w-1$ bits seguintes no vetor de bits de entrada, após a leitura desses w bits está função os converte da forma binária para decimal, e usa o valor obtido para consultar na tabela C os valores de excesso correspondentes aos bits analisados. Além da tabela C e da função *bitsRead*, em nossa implementação usamos outras duas funções que nos auxiliam no percurso em árvore, denominadas *leafInTree* e *numLeaf*, a primeira retorna o índice de um nó na rmM-tree correspondente ao número de uma folha, a segunda por sua vez retorna o número de uma folha correspondente à um nó da rmM-tree.

Exemplo 3: Tome com base agora, a nossa rmM-tree de exemplo, assuma $w = 2$, e suponha que queremos construir a folha de número 0 (nó 15), o intervalo que a folha 0 cobre vai de 0 à 4, assim precisaríamos ler os 4 bits que compõe esse intervalo, para um exemplo pequeno como este não há problemas em calcular esses valores iterativamente, mas imagine o que acontece com casos em que temos uma árvore de entrada maior, e um tamanho de blocos também maior, o processo se tornaria mais oneroso. Observe então como este processo é feito usando a tabela de excessos C.

- Como $w = 2$, convertemos primeiro os bits codificados em $BP[0,1]$
  $BP[0,1] = 11$ que em decimal corresponde à 3
  Fazemos então uma consulta na tabela C, na posição 3, que conforme explicamos anteriormente, traz os seguintes valores:

  $$C[3].e = 2; C[3].M = 2; C[3].m = 1; C[3].n = 1.$$

  Nesse momento, os valores de registro para nossa folha são:

  $$R[15].e = 2; R[15].M = 2; R[15].m = 1; R[15].n = 1.$$

- Agora precisamos computar os valroes correspondentes aos w bits restantes que compõe a nossa folha, temos que: $BP[3,4] = 11$ que em decimal corresponde à 3

  Agora que sabemos a que elemento em C essa parcela da nossa folha corresponde, podemos obter os valores de excesso para o intervalo completo da folha, imagine que estamos unindo dois intervalos coberto por um nó pai. Temos assim que:

  - $R[15].e = R[15].e + C[3].e = 2 + 2 = 4$
  - $R[15].M = max(R[15].M, R[15].e + C[3].M) = max(2, 2+2) = 4$
  - $R[15].m = min(R[15].m, R[15].e + C[3].m) = min(1, 2+1) = 1$

- $R[15].n = 1$.

Algoritmo 1: Construção da range min-Max tree binária

Input: Vetor de bits $BP[0, n-1]$, tabela de excessos C, tamanho de bloco b e de sub-bloco w, número de folhas r, quantidade de nós $nNodes$

```
1  // Construção dos nós folhas
2  for k ← 0 to r − 1 do
3      v ← leafInTree(k)
4      R[v].e ← 0; R[v].m ← −w; R[v].M ← w; R[v].n ← 0
5      for p ← (k ∗ (b/w)) + 1 to ((k + 1) ∗ b)/w do
6          x ← bistread((p − 1) ∗ w); if R[v].e + C[x].M > R[v].M then
             R[v].M ← R[v].e + C[x].M;
7          if R[v].e + C[x].m < R[v].m then R[v].m ← R[v].e + C[x].m;
8          else if R[v].e + C[x].m = R[v].m then
9              R[v].n ← R[v].n + C[x].n
10         end
11         R[v].e ← R[v].e + C[x].e
12     end
13 end
14 // Construção dos nós internos e raíz
15 for v ← nNodes − r − 1 to 0 do
16     vL ← (2 ∗ v) + 1; vR ← vL + 1
17     R[v].e ← R[vL].e + R[vR].e
18     R[v].M ← max(R[vL].M, R[vL].e + R[vR].M)
19     R[v].m ← min(R[vL].m, R[vL].e + R[vR].m)
20     if R[vL].m > R[vL].e + R[vR].m then R[v].n ← R[vR].n;
21     else R[v].n ← R[vL].n + R[vR].n;
22 end
```

2.5.2 Operações

As operações sobre a range min-Max tree são realizadas através de cálculos usando os valores de excessos definidos acima. Parte dessas já foram descritas nas Seções 2.3 e 2.4.1, como as operações $enclose$, $findclose$, $findopen$, $rank$ e $select$. As mesmas possibilitam a derivação de diversas outras operações, e portnato reduzem de modo eficiente a quantidade de estruturas necessárias para realziar diversas opeações de percurso em árvores. Mas além das operações mostradas, existem outras importantes operações suportadas pela rmM-tree, detalhamos algumas logo abaixo, e a tabela 2.1 mostra a lista completa das operações suportadas pela nossa implementação da range min-Max tree binária.

2.5.3 FwdSearch

O objetivo da operação *forward search* (busca à frente), é encontrar um excesso relativo d, em relação à um nó codificado por um índice i no vetor que representa uma árvor geral. Desse

modo, *fwdSearch* retorna um índice $j > i$, mais à esquerda possível de i, tal que, o nó definido por j está à uma profundidade d em relação ao nó codificado por i, o resultado dessa operação é dado pela expressão abaixo:

$$fwdsearch(i,d) = min\{j > i | excess(j) = excess(i) + d\}$$

Como veremos a seguir, dessa operação deriva-se diversas outras, tais como *findclose, lca (menor ancestral comum)* e outras operaçõess de percurso em árvore.

CORDOVA J. E NAVARRO (2016) descrevem o funcionamento dessa operação da seguinte maneira: dado um excesso desejado d, e um índice i, a partir do qual a busca dever ser feita, examinamos a folha k (com $k = \lceil i/b \rceil$) cujo o intervalo de cobertura engloba o índice i, a partir de $i+1$. Caso não encontremos d dentro desse intervalo, usamos os nós da rmM-tree para encontrar o bloco do vetor de entrada a qual d pertence. O processo para encontrar d neste caso deve ser feito do seguinte modo:

1. Defina uma variável dr, iniciada em zero, está variável será responsável por armazenar o excesso local de cada bloco lido. Quando a busca por d em um intervalo não obtiver sucesso, atualize o excesso computado até o momento, que é guardado em dr (basta adicionar à dr o campo de excesso referente ao nó inspecionado, seguindo as regras do ponto 2);

2. Verifique se o nó analisado é um filho à esquerda ou um filho à direita.

    (a) Caso o nó seja um filho à esquerda, verifique se d está contido no intervalo de excesso máximo e mínimo do seu irmão à direita ($dr + v.direita.m \leq d \leq dr + v.direita.M$); se essa condição não for satisfeita atualize o valor de dr (pelo ponto 1) e avance no percurso da rmM-tree através do pai do nó verificado;

    (b) Se o nó analisado for um filho à direita, simplesmente atualize o índice do nó corrente para pai de v, sem atualizar dr.

3. Prossiga recursivamente até que a condição $dr + v.direita.m \leq d \leq dr + v.direita.M$ seja cumprida, quando isso acontecer significa que encontramos o intervalo em que d está contido;

4. Afim de reduzir o intervalo de busca e encontrar o índice j de modo mais eficiente, iniciamos o percurso de descida em árvore através de $v.direita$ (esse processo reduzirá o tamanho de intervalo, sem excluir a resposta), seguindo pelo seu filho :

    (a) esquerdo, sempre que o excesso relativo d estiver no intervalo de contenção dos seus campos de excessos máximos e mínimo;

(b) direito, sempre que a condição anterior não for satisfeita, nesse caso precisamos atualizar o valor de dr pelo ponto a.

5. O processo descrito é repetido até que cheguemos a um nó folha, então procuramos pelo excesso relativo dr (atualizado ao longo do percurso) no bloco da folha em que nos encontramos. A primeira posição em que ocorrer este ecesso é a resposta para $fwdsearch$.

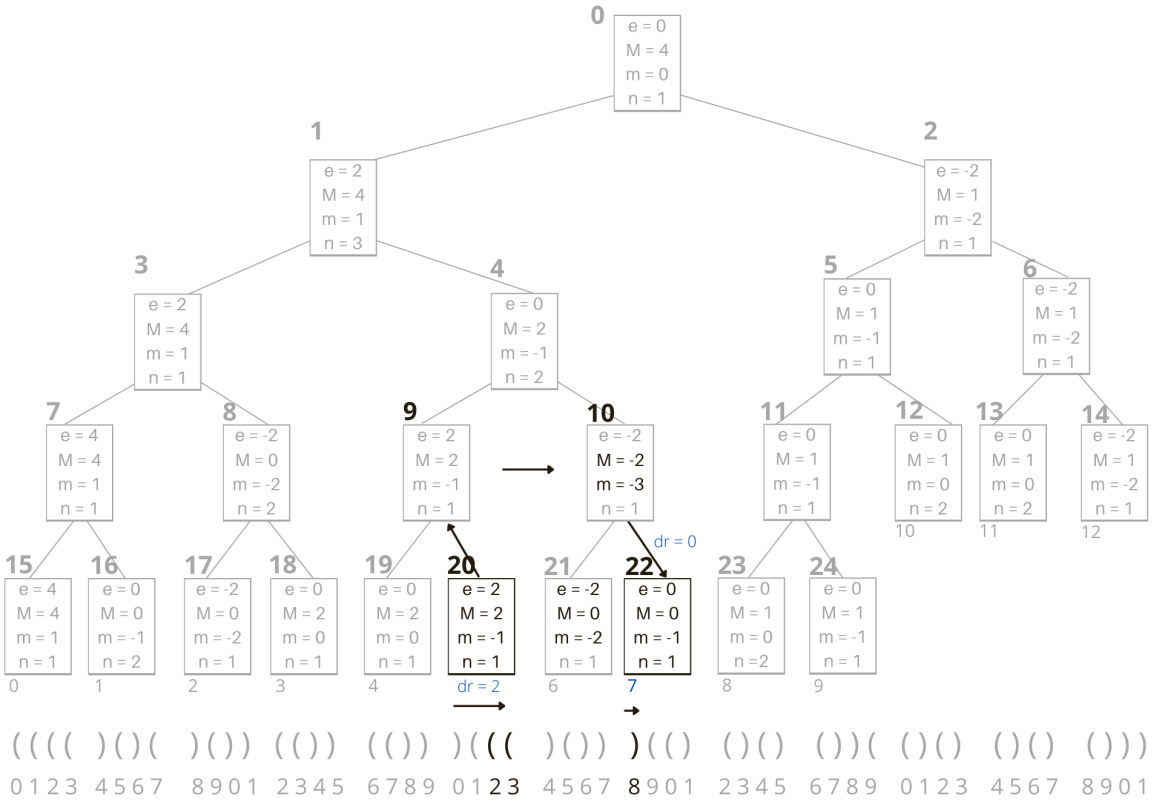
O exemplo 4 demonstra o processo acima, e o algoritmo 2 fornece mais detalhes de como todo esse processo é feito:

Examplo 4:

Dado um nó em BP, codificado em $i = 21$, encontrar o primeiro índice $j > i$,tal que $excess(j) - excess(i) = -1$.

A Figura 2.8, destaca os bits, registros e índices dos nós inspecionados.

Figura 2.8: Simulação da operação *fwdSearch(21,-1)* em uma rmM-tree binária.



O processo é iniciado através da inspeção dos bits cobertos pelo nó $v = 20$, os bits são inspecionados a partir da posição $i+1 = 22$, até chegarmos ao fim do intervalo deste nó que é $j = 23$, durante esse processo de inspeção simulamos a operação de $rank_($,

guardando em dr os valores computados, ao fim do intervalo, temos que $dr = 2$, portanto precisamos aumentar o nosso intervalo de possibilidades, e como dito, fazemos isso subindo na rmM-tree através dos seus nós, NAVARRO traz um passo interessante que otimiza a busca na árvore, que consiste em verificar o nó à direita do atual para verificar se a resposta se encontra nele, fazemo sesse processo olhando para os valores de excesso definidos no nó 20, sem obter sucesso. Atualizanmos então v para 9 (já sabemos que a resposta não está no nó 9), e verificamos se os campos do nó à direita (10) adicionados de dr, compreendem o excesso relativo buscado d, ou seja $2-3 \leq 1 \leq -2+2$, essa asserção é válida, portanto, setamos v como 10. Iniciamos a descida na árvore, e como buscamos a posição j mais à esquerda possível de i, verificamos primeiro se a resposta está contida no intervalo do filho esquerdo de nó 10, a asserção não é validada, e portanto adicionamos à dr o excesso local do nó 21 ($dr + (-2) = 0$), descemos então pelo filho direito de v, e nesse momento chegamos à um nó folha.

Com esse processo reduzimos ao máximo o intervalo a ser inspecionado, neste momento inicia-se a varredura bit-a-bit no intervalo coberto pelo nó 22. Nesse momento, temos que, $dr = 0$ e $BP[28] =)$, logo dr passe a valer -1, que é o exceso buscado inicialmente, e portanto a resposta para $fwdSearch(21,-1)$ é $j = 28$.

2.5.4 BwdSearch

Essa operação é bastante similar à *fwdSearch*, portanto não entraremos em tantos detalhes da mesma. Além disso mais à frente fornecemos um exemplo que esclarece o processo para esse tipo de busca. O ponto central de *bwdSearch* é que, ao invés de realizarmos uma busca "à frente", como na operação anterior, realizamos uma busca "para trás", assim *bwdSearch* busca por um índice $j < i$, mais à direita possível de i, tal que:

$$bwdsearch(i,d) = max\{j < i | excess(j) = excess(i) - d\}$$

Como explica NAVARRO, a relação acima impacta na nossa busca através da rmM-tree devido aos dados armazenados nas nossas estruturas serem assimétricos, o autor faz então as seguintes considerações relacionadas ao processo de inspeção na rmM-tree, usando *bwdSearch*:

- Buscamos por um excesso $j < i$, tal que:

$$excess(j) - excess(i) = -excess(j+1,i) = d,$$

  isso faz com que o índice i seja incluído na contagem;

- Queremos encontrar um índice j tal que o excesso relativo computado,dr, de i à j, seja igual ao excesso buscado, d. Mas como $j < i$ e portanto $dr = d = -excess(j+1,i)$,

Algoritmo 2: Busca por um excesso relativo d através de $fwdSearch(i,d)$.

Input: Índice i, a partir do qual a busca deve ser feita, excesso relativo d.
Output: Posição j onde ocorre d, ou $BP.size()$ caso a resposta não seja encontrada.

```
1  dr ← 0
2  j ← fwdBlock(i,d,&dr) // Algoritmo 3
3  if dr = d then return j;
4  k ← ⌊(i+1)/b⌋
5  v ← leafInTree(k)
6  while (v+1)&(v+2) and dr+R[v+1].m ≤ d ≤ dr+R[v+1].M do
7  |  if v mod 2 then dr ← dr+R[v+1].e ;
8  |  v ← ⌊(v−1)/2⌋
9  end
10 if (v+1)&(v+2) = 0 then return BP.size();
11 v ← v+1
12 while v < numberLeaves − 1 do
13 |  if dr+R[(2∗v)+1].m ≤ d ≤ dr+R[(2∗v)+1].M then v ← (2∗v)+1 ;
14 |  else
15 |  |  dr ← dr+R[(2∗v)+1].e
16 |  |  v ← (2∗v)+2
17 |  end
18 end
19 k ← numLeaf(v)
20 j ← fwdBlock((k∗b)−1,d,&dr)
21 if dr = d then return j;
22 else return BP.size();
```

devemos adicionar 1 unidade à dr ao inspecionarmos um bit codificado como 0, e diminuir dr em 1 unidade caso contrário;

- Os registros de excesso são computados a partir de um processo de inspeção de bits da esquerda para a direita. *bwdSearch*, realiza a pesquisa por excesso da direita para a esquerda, isso implica que o excesso buscado d, é encontrado em um nó somente quando a asserção $dr - R[v].e + R[v].m \leq d \leq dr - R[v].e + R[v].M$ for válida.

O algoritmo 4 detalha mais sobre esse processo, este algoritmo invoca um método denominado *bwdBlock*, este por usa vez é extremamente similar ao algoritmo 3 (a única ressalva é a questão de simetria discutida acima), portanto não fornecemos aqui *bwdBlock*[5].

Examplo 5: Dado um nó em BP, codificado em $i = 50$, encontrar o primeiro índice $j < i$,tal que $excess(j) - excess(i) = 0$

Como no exemplo anterior, a figura 2.9, destaca os bits e índices dos nós inspecionados para chegar a resposta esperada.

[5]Caso deseje ver como o mesmo é construído, ele está disponível em nosso repositório

Algoritmo 3: Busca pelo excesso relativo d em um nó folha, através de *fwd-Block(i,d&dr)*.

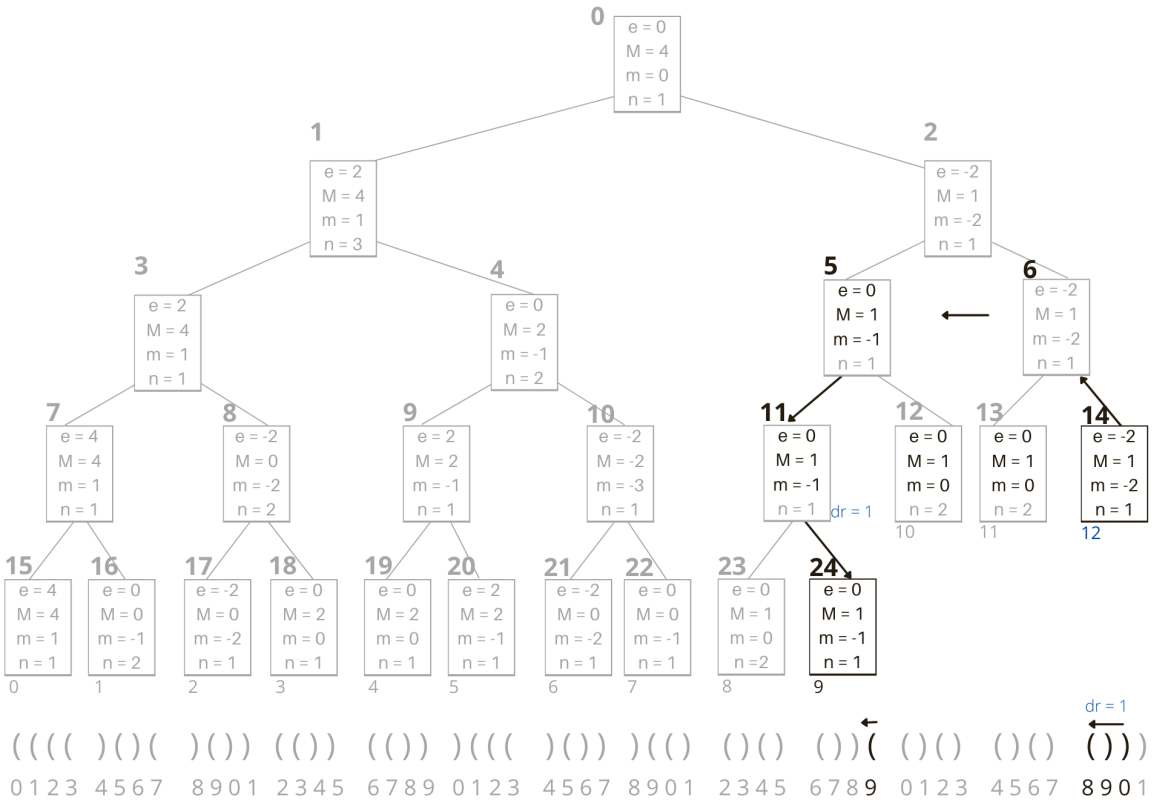
Input: Índice i a partir do qual começamos a busca, excesso buscado d, excesso computado até o momento $\&dr$.
Output: Posição j onde ocorre a resposta, ou $BP.size()$ caso a resposta não exista.

```
1  fb ← ⌈(i+1)/w⌉
2  lb ← ⌈(i+2)/b⌉ * (b/w)
3  for j ← i+1 to (lb*w)−1 do
4  |  if BP[j] = 1 then dr ← dr+1;
5  |  else dr ← dr−1;
6  |  if dr = d then return j;
7  end
8  for p ← fb+1 to lb do
9  |  x ← bitsRead((p−1)*w)
10 |  if dr+C[x].m ≤ d ≤ dr+C[x].M then break;
11 |  dr ← dr+C[x].e
12 end
13 if p > lb then return lb*b;
14 // dr não está no bloco subsequente
15 for j ← (p−1)*w to (p*w)−1 do
16 |  if BP[j] = 1 then dr ← dr+1;
17 |  else dr ← dr−1;
18 |  if dr = d then return j;
19 end
20 return BP.size()
```

A nossa busca inicia a partir da inspeção dos bits compreendidos no intervalo da folha (nó 14) que cobre o índice $i = 50$, aos inspecionarmos os bits que vão de $i = 50$ até $j = 49$, temos que $dr = 1$. Nesse momento dr ainda é diferente do excesso buscado d, fazemos uma verificação no nó à esquerda de v, e atestamos que o excesso procurado não está no intervalo do nó 13. Iniciamos a subida na árvore, atualizando v para 6, e antes de expandirmos o intervalo de busca navegando até o pai do nó v, analisamos o nó à esquerda de $v = 6$, verificando se a asserção[6] $1-0-1 \leq 0 \leq 1-0+1$ é válida, obtemos uma resposta positiva, e por isso interropemos o processo de subida na rmM-tree. Começamos a descidana árvore a partir de $v = 5$, verificamos se d está contido no intervalo de excesso do filho direito de v (lembrando, estamos buscando pelo j mais à esquerda de i), ou seja nó 12, sem obter sucesso, prosseguimos a busca atualizando v para o seu filho esquerdo. Verificamos agora se a asserção $dr - R[24].e + R[24].m \leq d \leq dr - R[24].e + R[24].m$ é válida para o filho direito do nó 11, e temos que a mesma é válida, como chegamos à um nó folha, interrompemos a busca pelos nós da rmM-tree, e iniciamos uma busca mais detalhada em $BP[39, 36]$

[6] $dr - R[v].e + R[v].m \leq d \leq dr - R[v].e + R[v].m$.

Figura 2.9: Simulação da operação $bwdSearch(50,0)$ em uma rmM-tree binária.



(de trás para frente), no índice 39, $dr = d$ e portanto a busca através de *bwdSearch* é encerrada retornando $j = 39$.

2.5.5 MinExcess

Esta é outra importante operação suportada pela rmM-tree. A partir dela podemos obter informações importantes sobre o nosso conjunto de dados, como o menor ancestral comum entre dois nós.

O objetivo da *minExcess*, é encontrar o excesso mínimo dentro de um intervalo i, j. Assim como para as outras operações, esta inicia-se a partir de uma inspeção bit-a-bit do intervalo que engloba o índice i, inspecionamos este intervalo fechado em i, até chegar ao fim do mesmo, ou até que cheguemos ao índice j. A cada cada bit inspecionado nesse processo, atualizamos o nosso excesso computado d, e verificamos se ele é o menor excesso computado até o momento (nesse caso atualizando o excesso mínimo computado). Terminando a inspeção deste bloco, caso j não esteja contido nele, iniciamos o processo de subida na árvore (caso j esteja contido, a busca é encerrada), nesse processo avaliamos se o excesso mínimo salvo em cada nó da rmM-tree é menor que o excesso mínimo computado até o momento, em caso afirmativo, atualizamos esse excesso mínimo, e assim como nas outras operações, independente da resposta, atualizamos o

Algoritmo 4: Busca por um excesso relativo d através de $bwdSearch(i,d)$

Input: Índice i a partir do qual a busca deve ser feita, excesso relativo d desejado
Output: Posição j onde ocorre o d, ou -1 caso a resposta não seja encontrada

```
1  dr ← 0
2  j ← bwdBlock(i,d,&dr) // Análogo ao algoritmo 3
3  if dr = d then return j;
4  k ← ⌊i/b⌋
5  v ← leafInTree(k)
6  while (v+1)&v and dr − R[v−1].e + R[v−1].m ≤ d ≤ dr − R[v].e + R[v−1].M do
7  |  if v mod 2 = 0 then dr ← dr − R[v−1].e ;
8  |  v ← ⌊(v−1)/2⌋
9  end
10 if (v+1)&v = 0 then return −1;
11 v ← v−1
12 while v < numberLeaves − 1 do
13 |  if dr − R[(2∗v)+2].e + R[(2∗v)+2].m ≤ d ≤
   |    dr − R[(2∗v)+2].e + R[(2∗v)+2].M then
14 |  |  v ← (2∗v)+2
15 |  end
16 |  else
17 |  |  dr ← dr − R[(2∗v)+2].e
18 |  |  v ← (2∗v)+1
19 |  end
20 end
21 k ← numLeaf(v)
22 j ← bwdBlock(((k+1)∗b)−1,d,&dr)
23 if dr = d then return j;
24 else return −1;
```

excesso computado até o momento. Ressalta-se que o percurso em árvore para esta operação é idêntico ao percurso em árvore para *fwdSearch*, sendo interrompido, no momento em que estamos prestes a inspecionar um nó v, que seja ancestral do nó folha cujo intervalo engloba o índice j. Para tanto, precisamos de algumas evidências, trazidas por NAVARRO, são elas:

- A profundidade de qualquer nó em $R[v]$ é dada por $\lfloor \log v \rfloor + 1$;
- O pai de um nó, é dado por $\lfloor v/2 \rfloor$, logo u é ancestral de um nó v, se e somente se a asserção abaixo for válida,

$$\lfloor v/2^{\lfloor logv \rfloor - \lfloor logu \rfloor} \rfloor = v$$

- Conforme explica o autor, a asserção acima pode falhar quando o nó u for um nó folha, e tiver uma profundidade maior que a do nó alvo (a folha que contém o índice j), de acordo com o autor isso pode ocorrer quando o número de folhas não for uma

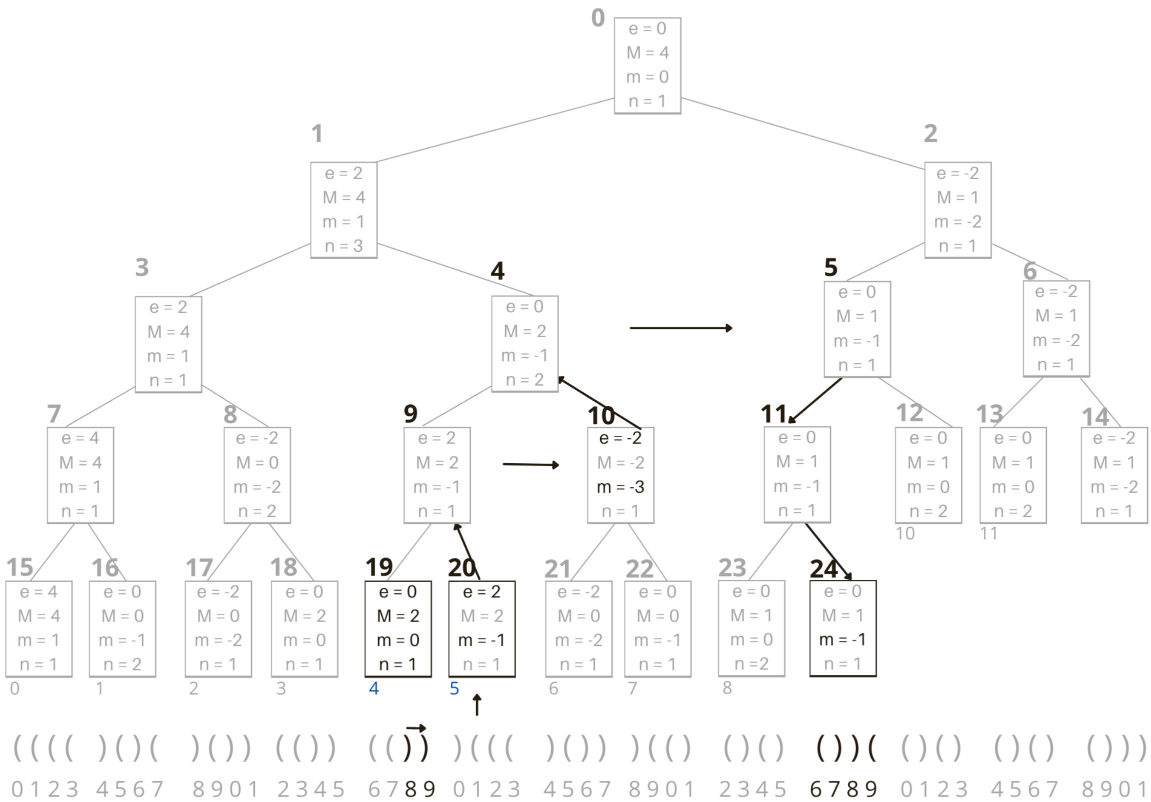
potência de 2. Para suprir esse problema devemos criar outro critério de continuação de subida na árvore, que é verificar se o índice do nó inspecionado é ou não, maior do que o índice do nó alvo, em caso afirmativo a busca prossegue com a subida na árvore.

Ao encontrarmos o ancestral do nó alvo, interrompemos o processo de subida na árvore, e começamos o processo de descida a partir deste ancestral, esse processo ocorre de modo similar aos outros, verificamos primeiro se o filho esquerdo do nó visitado é ancestral do nó alvo. Em caso negativo usamos os valores de excesso deste nó para verificar se existe um excesso mínimo menor que o computado até o momento, atualizamos esse valor conforme a resposta que obtivermos, e atualizamos o nosso nó corrente para o filho à direita deste nó. Em caso positivo, atualizamos o nó corrente, para o seu filho esquerdo sem verificar os valores de excesso. Repetimos todo esse processo até que cheguemos a um nó folha, a partir daí fazemos uma inspeção bit-a-bit, até chegar na posição j, e finalmente encerrando a busca tendo o menor excesso computado no intervalo $BP[i, j]$.

O exemplo a seguir ilustra como o processo dessa operação é feito.

Examplo 6: Dado o intervalo $BP[18, 39]$, retornar o menor excesso.

Figura 2.10: Simulação da operação $minExcess(18, 39)$ em uma rmM-tree binária.



Varremos primeiramente o intervalo que vai de $i = 18$ até $p = 19$, nesse momento temos que o excesso mínimo (m) computado é -2, e o excesso local (d) é também -2. Como j não está incluso no intervalo $18, 19$, precisamos iniciar o processo de subida na árvore, antes disso identificamos o nó folha que cobre o intervalo de j, este nó é dado por $l = 24$, verificamos, então se o excessmo mínimo no nó à direita de $v = 19$, é menor que o excesso computado até o momento, como a asserção é válida $-2-1 < -2$[7], atualizamos o valor de m para -3, e o valor de d para 0. Subimos na árvore pelo nó 9, e analisamos se o nó à direita, $v = 10$, é ancestral de l, e obtivemos uma negativa, atualizamos o valor de d para -2, e subimos na árvore através do nó 4. Novamente, verificamos se o nó à direita (5) do nó atual é ancestral do nó $l = 24$, e nesse momento temos uma assertiva validada. Interrompemos e partir daqui iniciamos o processo de descida pela rmM-tree, a partir de $v = 5$, como o filho esquerdo (nó 11) de v, é ancestral do nó 24, atualizamos v para 11. Em seguida, verificamos que o nó 23 (filho esquerdo), não é ancestral do nó $l = 24$, prosseguimos então para o filho direito, que é um nó folha e o próprio, nó 24. Pecorremos os bits do intervalo coberto pelo nó l, até chegarmos à j, e, terminando a inspeção temos que o excesso mínimo em $BP[18, 39] = -3$.

Uma outra variação dessa operação é a *maxExcess*, o processo para responder está é completamente análogo, ao descrito acima, mudando apenas os registros da rmM-tree analisados, ao invés de verificarmos o mínimo, verificamos o máximo.

2.5.6 MinCount e MinSelectExcess

Essas operações possuem bastate similaridade entre si, e também são análogas à *minExcess*, portanto não entraremos em detalhes sobre as duas, novamente, ambas estão disponíveis em nosso repositório.

De modo geral, o objetivo da operação *minCount* é computar o número de vezes que o excesso mínimo aparece no intervalo $BP[i, j]$, ao passo que a operação *minSelectExcess*, objetiva encontrar o índice dentro do intervalo i, j, onde ocorre a $t-$ésima ocorrência do excesso mínimo computado neste mesmo intervalo. Observe que para ambas as operações, é necessário primeiro computar o excesso mínimo dentro do intervalo fornecido, para tanto, podemos fazer uma chamada a função *minExcess*, e então pecorrer a rmM-tree em busca da resposta esperada para cada operação.

Para operação *minCount*, criamos um acumulador que é incrementado através do campo n, da rmM-tree, sempre que passarmos por um nó cujo o campo de excesso mínimo for igual ao excesso computado por *minExcess*. A operação *minSelectExcess* segue pelo mesmo caminho, criamos um acumulador que recebe o valor t passado para a função, e a diferença consiste no fato de que ao encontrarmos um nó cujo o campo de excesso mínimo é igual ao valor computado

[7] $d + R[20].m < d]$

por *minExcess*, subtraímos do acumulador o valor armazenado em n, a resposta nesse caso é encontrada quando o acumulador atingir a marca 0. Para ambas as operações, o processo de subida e descida na árvore se dá de igual modo ao mostrado em *minExcess*.

Essas são as únicas operações que usam o valor n da rmM-tree, portanto não há necessidade de armazenar este, caso o objetivo da sua estrutura não englobe uma destas operações ou suas derivadas.

2.5.7 Derivadas

Esta seção demonstra como diversas operações sobre a rmM-tree podem ser escritas de modo eficiente a partir das operações descritas anteriormente.

- **rmq**: busca pela primeira posição onde ocorre o excesso mínimo computado dentro do intervalo $BP[i, j]$. Ou seja:

$$rmq(i, j) = \min\{\arg\min\{excess(p) | i \leq p \leq j\}\}$$

  Para à responder esta operação, podemos computar primeiro o exesso mínimo dentro do intervalo i, j, e em seguida realizar uma busca a partir de i. Temos assim:

$$rmq(i, j) = fwdSearch(i-1, minExcess(i, j))$$

  Repare que o parâmetro do intervalo em *fwdSearch* é subtraído em 1, isso acontece porque o intervalo de busca define *fwdSearch* é aberto, ou seja, não incluí o índice passado nas buscas, ao passo que o intervalo de *rmq* é fechado, devendo incluir i e j nas buscas.

- **rMq**: similar a operação anterior, *rMq*, busca a posição mais à esquerda de i onde ocorre o excesso máximo computado em i, j. Ou seja:

$$rMq(i, j) = \max\{\arg\max\{excess(p) | i \leq p \leq j\}\}$$

  Assim:

$$rMq(i, j) = fwdSearch(i-1, maxExcess(i, j))$$

- **findClose**: essa operação recebe como parâmetro um índice i, correspondente à um de abertura, e a partir da operação *fwdSearch*, busca o parênteses de fechamento correspondente. *fwdSearch* irá fazer a varredura a partir de $i+1$ até encontrar o índice $j > i$, tal que $excess(j) - excess(i+1) = -1$. Assim:

$$findClose(i) = fwdSearch(i, -1)$$

- **findOpen**: dado um índice, tal que $BP[i] = 0$, *findOpen*, faz uma busca por um $j < i$ tal que $excess(j) - excess(i) = 0$ e $excess(j) + 1 = 1$. Isso nos dá:

$$findOpen(i) = bwdSearch(i, 0) + 1$$

- **enclose**: dado um índice i, que indica a codificação de um nó x, *enclose* usará *bwdSearch* para encontrar a posição $j < i$ mais à direita de i, tal que $excess(j) - 2 = 1$. Ou seja:

$$enclose(i) = bwdSearch(i, -2) + 1$$

- **levelAncestor**: essa função buscará pelo nó y que esteja d níveis acima de um nó x, para tanto podemos usar *bwdSearch*, por questões de simetria, já discutidas anteriormente, devemos setar d como negativo em *bwdSearch*, e assim temos:

$$levelAncestor(x, d) = bwdSearch(x, -d - 1) + 1$$

- **isAncestor**: essa função tem por objetivo verificar se um nó x é ancestral de um nó y, para tanto basta verificar se y está contido na hierarquia de x, podemos para tanto usar *findClose*, do seguinte modo:

$$isAncestor(x, y) = (x < y \text{ and } y < findClose(x))$$

- **parent**: dado um índice i, correspondente ao bit que codifica um nó x, *parent* busca através de *enclose* o índice em BP do nó que codifica o nó pai de x:

$$parent(i) = enclose(i)$$

- **lca**: busca o menor ancestral comum entre dois nós, codificados nos índices i e j. Existem 3 possibilidades de retorno para *lca*, são eles:

$$lca(i, j) = \begin{cases} i, \text{se } isancestor(i, j); \\ j, \text{se } isancestor(j, i); \\ parent(rmq(i, j) + 1) \end{cases}$$

- **deepestNode**: dado um nó em BP, *deepestNode* retorna o filho com maior profundidade deste nó (localizado mais à esquerda possível). Sabemos que para calcular o maior excesso possível a partir de i podemos usar a operação *maxExcess*, como esse excesso deve estar limitado ao escopo do nó codificado em i, temos a seguinte relação:

$$deepestNode(i) = rMq(i, findClose(i))$$

A operação *rMq* é usada aqui para invocar *maxExcess* e em seguida retornar a posição exata onde ocorre o excesso computado.

- **degree**: esta operação contabiliza o número de filhos de um nó codificado em i, para simular ela, basta obter a quantidade de vezes que atingimos o excesso mínimo em $BP[i, findClose(i)]$, excluindo os limites que definem este nó, ou seja:

$$degree(i) = minCount(i+1, findClose(i)-1)$$

- **childRank**: dado um índice i, *childRank*, contabiliza o número de irmãos que o nó codificado em i possuí à sua esquerda, tendo as operações definidas anteriormente, podemos responder à esta operação do seguinte modo:

$$childRank(i) = minCount(parent(i)+1, i)+1$$

- **nextSibling**: dado um nó codificado em i, *nextSibling* busca pelo nó codificado em j, tal que $j > i$.

$$nextSibling(i) = findClose(i)+1$$

- **prevSibling**: análaga a *nextSibling*, esta operação busca pelo irmão imediatamente à esquerda do nó codificado por i em BP. Ou seja:

$$prevSibling(i) = findOpen(i-1)$$

- **child**: dado um índice i que codifica um nó, *child* calcula a posição em *BP* onde ocorre a codificação do $t-$ésimo filho de i. Ou seja, ela calcula o índice onde ocorre pela $t-$ésima vez o excesso mínimo dentro do intervalo do nó i, desse modo, podemos escrever *child* em função de *findClose* e minSelectExcess:

$$p = findClose(i)$$

$$child(i) = minSelectExcess(i+1, p-1, t-1)+1$$

- **lastChild**: busca pelo último filho de um nó codificado em um índice i:

$$lastChild(i) = findOpen(findClose(i)-1)$$

- **subtreeSize**: calcula o tamanho da subárvore enraízada no nó i, para tanto, basta

realizar uma contagem dos nós codificados no intervalo do nó i, ou seja:

$$subtreeSize(i) = \lfloor (findclose(i) - i + 1)/2 \rfloor$$

- **levelNext**: busca o nó mais a direita do nó codificado por i, em seu mesmo nível:

$$levelNext(i) = fwdSearch(findClose(i), 1)$$

- **levelPrev**: análaga a *levelNext*, esta função busca pelo primeiro nó a esquerda do nó codificado em i, que possuí a mesma profundidade deste. Como estamos falando de uma busca à esquerda de um índice, usamos *bwdSearch* para realizar está operação:

$$levelPrev(i) = findOpen(bwdSearch(i, 0) + 1)$$

- **levelLeftMost**: busca pelo nó mais à esquerda da raíz, com profundidade d, assim:

$$levelLeftMost(d) = fwdSearch(0, d - 1)$$

- **levelRightMost**: usando como referência a raíz, busca pelo nó mais à direita, cuja profundidade é igual à d:

$$levelRightMost = findOpen(bwdSearch(BP.size() - 1, d) + 1)$$

Algumas das operações a seguir, além das nossas primitivas, usam em seu escopo as operações suportadas pela estrutura de vetores de bits (algumas, como o caso de *depth* usam apenas estás últimas). Isso nos mostra também a importância do cuidado ao escolhermos uma estrutura de suporte apropriada afim de acelerar nossas operações:

- **firstChild**: retorna o primeiro filho do nó codificado por i, logo:

$$firstChild(i) = i + 1$$

- **isLeaf**: dado um nó codficiado em i, verifica se este é um nó folha ou não, essa é uma das operações mais simples da nossa estrutura e necessista apenas de 2 comparações, veja abaixo:

$$isLeaf(i) = (BP[i] == 1 \text{ and } BP[i + 1] == 0)$$

- **depth**: computa a profundidade de um nó codficado por i, para tanto basta verificar o exesso em $BP[0, i]$, ou seja:

$$depth(i) = excess(i)$$

- **leafRank**: contabiliza a quantidade de folhas existente em um intervalo que vai de 0 à i. Em nossa implementação, usamos a operação *rank*, suportada pela estrutura de vetores de bits. Assim, basta buscar a quantidade de ocorrêncas de bits 1 seguidos por bit $0's$.

$$leafRank(i) = rank_{10}(i)$$

- **leafSelect**: esta operação nos permite buscar pela $i-$ésima folha em BP, para essa operação buscamos pela $i-$ésima ocorrência do padrão 10:

$$leafSelect(i) = select_{10}(i)$$

- **leftMostLeaf**: retorna o índice da folha localizada mais à esquerda do nó codificado em i.

$$leftMostLeaf(i) = leafSelect(leafRank(x-1)+1)$$

- **rightMostLeaf**: busca a folha mais à direita do nó codificado em i:

$$rightMostLeaf(i) = leafSelect(leafRank(findClose(i)))$$

- **preRank**: calcula a quantidade de ancestrais de um nó codificado em i.

$$preRank(i) = rank_1(i)$$

- **postRank**: calcula a quantidade de ancestrais do nó i somados aos seux x filhos.

$$postRank(i) = rank_0(findClose(i))$$

- **preSelect**: retorna a posição do $i-$ésimo nó, usando um percurso pré-ordem:

$$preSelect(i) = select_1(i)$$

- **postSelect**: retorna a posição do $i-$ésimo nó, usando um percuso pós-ordem, nesse caso podemos usar *findOpen* que é derivada da nossa primitiva *bwdSearch*:

$$postSelect(i) = findOpen(select_0(i))$$

2.6 rmM-tree k-ária

Apesar da range min-Max tree possuir um desempenho satisfatório, obedecendo ao limite de espaço definido pela teoria informação e oferecendo suporte à diversas operações em

Tabela 2.1: Operações suportadas pela rmM-tree bináriaW

Operação	Retorno
fwdSearch(i,d)	Índice j, tal que $excess(j) = excess(i) + d$
bwdSearch(i,d)	Índice j, tal que $excess(j+1,i) = -d$
minExcess(i,j) / maxExcess(i,j)	Excesso mínimo/máximo em i, j
minCount(i,j)	Número de vezes que o excesso mínimo aparece em i, j
minSelectExcess(i,j,t)	Índice da $t-$ésima ocorrência do excesso mínimo em um intervalo
enclose(i)	Posição do parênteses de abertura que envolve $BV[i]$
rmq(i,j) / rMq(i,j)	$p >= i$ mais à esquerda de i, onde ocorre o excesso mínimo/máximo do intervalo dado
$rank_1$(i) / $rank_0$(i)	Número de parênteses abrindo/fechando em $BV[0,i]$
$select_1$(i) / $select_0$(i)	Posição do i-ésimo parênteses de abertura/fechamento
preRank(i)/postRank(i)	rank de i calculado a partir de um percurso *preorder ou postorder*
preSelect(i)/postSelect(i)	retorna o nó com *preorder/postorder* i
isLeaf(i)	Verifica se $BV[i]$ codifica uma folha
isAncestor(i,j)	Verifica se o nó codificado em i é ancestral de j
depth(i)	Profundidade do nó i
parent(i)	Obtém o pai do nó i
firstChild(i) / lastChild(i)	Retorna o primeiro/último filho do nó codficiado em $BV[i]$
child(i,t)	$t-$ésimo filho do nó codificado em i
nextSibling(i) / prevSibling(i)	Primeiro irmão à direita/esquerda de i
subtreeSize(i)	Número de nós enraízados na subárvore de i
levelAncestor(i,d)	Ancestral j de i tal que $depth(j) = depth(i) - d$
levelNext(i) / levelPrev(i)	Nó à direita/esquerda de i com a mesma profundidade de i.
levelLeftMost(d) / levelRightMost(d)	Nó mais à esquerda/direita, com profundidade d.
lca(i,j)	Menor ancestral comum dos nós codificados em i e j
deepestNode(i)	Nó mais profundo de i (mais à direita possível)
degree(i)	Número de filhos do nó i
childRank(i)	Número de irmãos à esquerda do nó codificado em i
leafRank(i)	Número de folhas à esquerda da folha codificada em i
leafSelect(i)	$i-$ésima folha em $BV[0, size-1]$
leftMostLeaf(i)	folha codificada em j, mais à direita de i, tal que $j < i$
rightMostLeaf(i)	folha codificada em j, mais à direita possível, tal que $j \leq i$

tempo eficiente, quando se trabalha com grandes conjuntos de dados a mesma pode realizar muitas transferências entre cache e memória RAM. Isso se deve ao fato de que a rmM-tree é uma árvore construída no formato de árvore binária, o que implica em um fator de ramificação baixo, acarretando em uma perda na localidade espacial durante o processo de navegação na árvore.

Como vimos em outros trabalhos, aumentando o número de ramificações de uma árvore, podemos sanar os problemas acima citados. Nosso objetivo portanto com este trabalho é propor um tipo de rmM-tree com um número maior de ramificações, a proposta para essa estrutura será melhor detalhada no capítulo a seguir.

3

Range min-Max tree k-ária

Esta capítulo refere-se ao objetivo central deste trabalho, apresentando a proposta de otimização da estrutura criada por SADAKANE K. E NAVARRO (2010). Mostraremos primeiro no que consiste essa adaptação, após isso definiremos a forma como serão compostos os nós da rmM-tree e como a mesma deve ser construída, apresentaremos também as operações suportadas por esta estrutura, detalhando as princpipais delas. Por fim, destaca-se que os exemplos usados como base neste capítulo, serão os mesmos exemplos usado no capítulo de apresentação da rmM-tree clássica, afim de facilitar a compreensão e comparação das duas propostas.

3.1 Visão Geral

Para este trabalho buscamos unir algumas características das árvores B com a range min-Max tree. Com até m filhos, as árvores B possuem altura igual a $\log_m n$ e alto fator de ramificação, essas características combinadas a Range min-Max tree, que é uma estrutura compacta, e portanto cabe na memória principal, possibilitará um uso otimizado de cache. Pois o alto fator de ramificação da estrutura, implicará em um uso satisfatório do príncipio de localidade de dados, e em decorrência disto em uma diminuição do número de cache misses (RAO J. E ROSS, 2000).

Ademais, como também mostram HANKINS R. A. E PATEL (2003) o número de faltas de cache é limitado pela altura da árvore, o que como já expomos é menor para as árvores B, espera-se assim reduzir o tempo para as operações sobre a rmM-tree k-ária, se comparada ao tempo gasto pelas operações na rmM-tree binária. Por último, vale ressaltar que a complexidade assintótica das duas estruturas será a mesma, porém devido ao número reduzido de transferência de dados entre cache e memória RAM, a constante oculta desse limite assintótico será reduzida (CORMEN, 2012), melhorando o desempenho do sistema.

Assim, dado uma representação compacta de uma árvore geral T, na forma de parênteses balanceados, ou vetores de bits, é escolhido um tamanho de bloco b, esse tamanho de bloco, assim como na estrutura clássica define a cobertura de um determinado intervalo. Na proposta original (SADAKANE K. E NAVARRO, 2010), cada nó da Range min-Max tree é responsável pela cobertura de um desses intervalos. Nesta adaptação propõe-se o agrupamento múltiplo de

intervalos no formato de chaves em nós.

Com base nisso, após definir uma ordem m, com m maior que 2, para a rmM-tree, armazenaremos m chaves [1], contendo intervalos de máximos e mínimos de tamanho b por nó da nossa estrutura. Isso implica que teremos até m^2 áreas cobertas por nó (pois m filhos $\cdot (m)$ chaves), dessa maneira temos o exposto: se na versão clássica da rmM-tree cada nó folha armazena no máximo um intervalo, e portanto possuí $r = \lceil n/b \rceil$ folhas, na versão k-ária teremos $r = \lceil n/(b \cdot m) \rceil$ folhas.

Como a altura de uma árvore pode ser calculada a partir de $h = \lceil \log_m r \rceil$ essa simples mudança reduzirá drasticamente a altura da nossa estrutura, quando comparada a rmM-tree clássica, fazendo com que o número de eventuais transferências de dados entre os níveis da memória torne-se menor.

Tomemos como exemplo um tamanho de bloco $b = 32$, suponha um vetor de bits de tamanho igual à 4.675.776.358, e que a nossa rmM-tree tem ordem 16, o exemplo logo abaixo ajuda a esclarecer o expostos acima, mostrando a altura, quantidade de folhas de cada versão da rmM-tree.

$$r = \lceil 4.675.776.358/32 \rceil = 146.118.012$$
$$h = \lceil \log_2(146.118.012) \rceil = 28, \text{ para a estrutura clássica.}$$

$$r = \lceil (4.675.776.358/(32 \cdot 16)) \rceil = 9.132.376$$
$$h = \lceil \log_{16}(4.711.244) \rceil = 6, \text{ para a estrutura otimizada.}$$

No pior caso do exemplo acima, a rmM-tree clássica terá 28 transferências de dados entre os níveis da hierarquia de memória, ao passo que no pior caso da rmM-tree otimizada teremos 6 transferências. Isso acontece porque ao aumentar o número de dados cobertos por nós, tiramos proveito do príncipio de localidade espacial (HENNESSY J. L. E PATTERSON, 2014). Assim a estrutura da nossa árvore ao maximizar o volume de dados enviados a cada transferência, faz com que os mesmo sejam referenciados de modo mais rápido.

3.2 Registros

Os registros da range min-Max tree k-ária permanecem exatamente os mesmos, sendo eles: *excesso local (e)*, que indica o excesso total computado em um intervalo; *excesso local mínimo (m)* que é relativo ao menor exesso computado a cada posição do intervalo $[s,e]$; *excesso local máximo (M)* que é análogo ao excesso mínimo; e *número de vezes que o excesso mínimo ocorre (n)* em um intervalo. A grande diferença, é que agora esses valores de excessos estão

[1] Uma árvore B armazena até m-1 chaves por página, aqui optamos por manter m chave, por nó, afim de garantir uma cobertura completa dos intervalos à cada nível

agrupados em até m blocos de tamanho b por folha, isso nos dá às seguintes relações para os nós internos e raíz:

- $R[v][k].e = \sum_{i=0}^{l-1} R[j][i].e$
- $R[v][k].m = min(R[j][0].m, ..., R[j][0].e + ... + R[j][l-1].e + R[j][l].m)$
- $R[v][k].M = Max(R[j][0].M, ..., R[j][0].e + ... + R[j][l-1].e + R[j][l].M)$
- $R[v][k].n$ o valor desse registro irá depender da quantidade de vezes que o excesso mínimo computado aparece entre as chaves do nó j. Caso esse valor não se repita, o valor de n, em v, será definido pelo valor de n na chave que contém o excesso mínimo, caso contrário, $R[v][k].n$ será dado pelo somatório de n, nas chaves do nó j onde o excesso mínimo se repete.

onde:

- k, é a $k-$ésima chave de v e aponta para o nó $(v*m)+1+k$ (com $0 \leq k < m$);
- j é o $j-$ésimo filho de v;
- l é a quantidade de chaves em j.

Examplo 7: Usando o mesmo exemplo de sequência de parênteses balanceados da seção 2.5, assuma a cobertura de bloco igual à 4, e uama rmM-tree de ordem também igual à 4. Temos assim que a Range min-Max tree terá 4 nós folhas, e no máximo 4 chaves por nó, pois:

$$r = \lceil n/(b \cdot m) \rceil \rightarrow \lceil (52/16) \rceil = 4$$

$$h = \lceil \log_m r \rceil \rightarrow \lceil \log_4 4 \rceil = 1$$

Como nesta adaptação as chaves da rmM-tree assumem o papel dos blocos da implementação clássica, as chaves dos nós folhas serão idênticas às folhas definidas no exemplo 1, em ordem. Iremos portanto nos concentrar em exemplificar a forma como é feito o cálculo de 1 das chaves de um dos nós superiores da nossa estrutura, que devido às características da nossa árvore base, será o nó raíz:

Tomemos a chave de número 2, do nó raíz da nossa estrutura, o nó ao qual está chave se refere, é o nó $v = (4*0)+1+2 \therefore v = 3$, ou seja, temos que os valores para as chaves deste nósão:

- Chave 0:
  Cobertura: $BP[32,35], e = 0, M = 1, m = 0, n = 2$

- Chave 1:
  Cobertura: $BP[36,39], e = 0, M = 1, m = -1, n = 1$
- Chave 2:
  Cobertura: $BP[40,43], e = 0, M = 1, m = 0, n = 2$
- Chave 3:
  Cobertura: $BP[43,47], e = 0, M = 1, m = 0, n = 2$

Assim, a chave 2, da página raíz, da qual descende está folha terá os seguintes valores:

- $R[0].keys[2].e = R[3].keys[0].e + R[3].keys[1].e + R[3].keys[2].e + R[3].keys[3].e = 0$
- $R[0].keys[2].M = 1$, pois:

$$\begin{aligned} R[0].keys[2].M = max(&R[3].keys[0].M, \\ &R[3].keys[0].e + R[3].keys[1].M, \\ &R[3].keys[0].e + R[3].keys[1].e + R[3].keys[2].M, \\ &R[3].keys[0].e + R[3].keys[1].e + R[3].keys[2].e + R[3].keys[3].M) \\ &max(1,1,1,1) = 1 \end{aligned}$$

- $R[0].keys[2].m = -1$, pois:

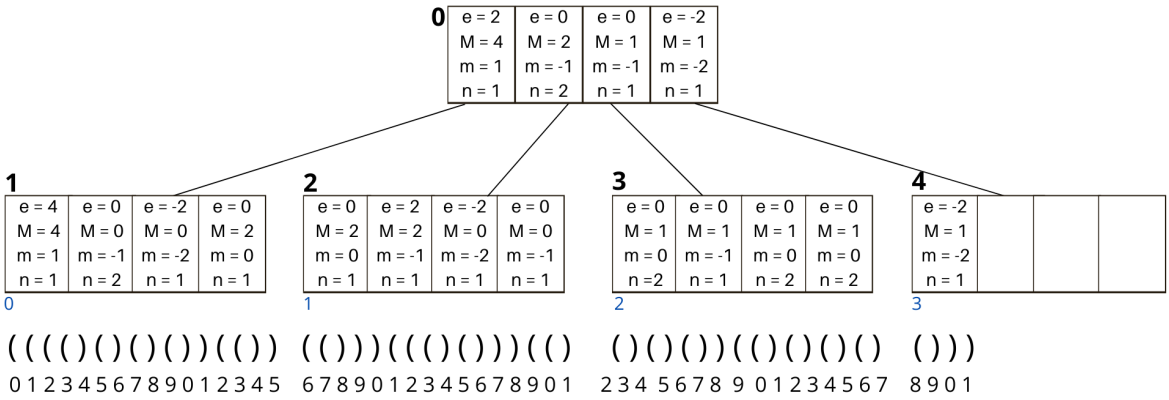
$$\begin{aligned} R[0].keys[2].M = min(&R[3].keys[0].m, \\ &R[3].keys[0].e + R[3].keys[1].m, \\ &R[3].keys[0].e + R[3].keys[1].e + R[3].keys[2].m, \\ &R[3].keys[0].e + R[3].keys[1].e + R[3].keys[2].e + R[3].keys[3].m) \\ &min(0,-1,0,0) = -1 \end{aligned}$$

- $R[0].keys[2].n = 1$, pois o excesso ocorre uma única vez, na chave de número 1 do nó 3 (folha 2).

Tendo computado os valores de excesso máximo e mínimo para todas as chaves dos nós folhas, e posteriormente para a raíz, temos a rmM-tree da Figura 3.1:

O processo de construção da rmM-tree k-ária é similar ao processo de construção da rmM-tree binária descrito no capítulo 2, a diferença é que, para este caso precisaremos guardar em um nó até m registros de excesso. Podemos para tanto fazer o uso de uma estrutura auxiliar como uma lista ou um vetor (presente em linguagens de programação como C++), onde cada elemento dessa estrutura representará as chaves, de um nó v da rmM-tree, ao qual essa estrutura

Figura 3.1: rmM-tree 4-ária, com tamanho de bloco igual à 4. No canto superior esquerdo de cada nó, em negrito, é mostrado o índice do mesmo na rmM-tree, já no canto inferior esquerdo de cada folha, em azul, vemos a ordem da mesma.



auxiliar está associada. Veja como isso pode ser feito usando o algoritmo 5. A tabela C, as funções *bitsread, leafInTree* e *numLeaf*, algumas citadas nesse e outras em outros algoritmos seguem a mesma abordagem da rmM-tree clássica.

3.3 Operações

Para este trabalho a nossa estrutura fornece suporte às duas principais primitivas da Range min-Max tree clássica, FwdSearch e BwdSearch (além daquelas já suportadas pela estrutura de vetores de bits). Além destas duas, nossa árvore de intervalos máximos e mínimos, fornece suporte à 5 operações derivadas daquelas apoiadas pela estrutura de vetores de bits (sobre a qual a nossa árvore é construída), e outras 16 operações, que são derivadas das duas primitivas exploradas adiante.

3.3.1 FwdSearch

Como já dito no capítulo anterior a operação foward search, realiza uma busca a partir de um índice i, afim de encontrar a posição j, onde ocorre um excesso relativo d.

Essa operação, nessa versão da rmM-tree, se dá parcialmente de igual forma ao do modo clássico, varremos primeiro o bloco da folha, de w em w bits em busca do excesso desejado, a partir de $i+1$. A diferença é que agora precisamos verificar por até m blocos de tamanho b dentro dos nós da nossa estrutura. Para tanto, acrescentamos à fwdSearch em sua versão clássica duas outras funções, são elas fwdKey e fwdVerifySibling.

A primeira função é chamada sempre que estamos visitando um nó folha, ou seja no ínicio da busca e ao final da mesma, quando já encontramos o nó alvo que contém a resposta. FwdKey tem por objetivo verificar as chaves do nó visitado, em busca daquela cujo os campos de intervalo máximo e mínimo compreendem o excesso relativo d. O processo é muito similar à verificação de um nó, a cada chave pecorrida verificamos se a asserção

$dr + R[v].key[l].m \leq d \leq dr + R[v].key[l].M$, caso a asserção seja significa que encontramos o intervalo que contém a resposta desejada, então nos encaminhamos para a função fwdBlock algoritmo 3, que fará uma varredura bit-a-bit da chave que contém a resposta. Se a asserção acima for falsa, adicionamos a dr o excesso local da chave que acabamos de visitar, e prosseguimos para a próxima chave. O algoritmo 6 fornece mais detalhes de como esse processo de busca por excesso nas folhas funciona.

Caso o excesso d não seja encontrado na folha, acionamos o processo de subida na árvore, aliado à nova função que mencionamos, *fwdVerifySibling*. O objetivo dessa função é similar ao proposto por NAVARRO (2016), quando o autor sugere que em cada nível verifiquemos se o excesso procurado está no irmão do nó atual. Assim *fwdVerifySibling* calcula a quantidade de irmãos do nó v existentes à sua direita - calcula-se o índice do primeiro filho do pai de v, e subtraímos de v o resultado, o número de irmãos à direita de v é o número chaves do seu pai, subtraído desse resultado - e então pecorremos as chaves de cada irmão à direita de v, em busca do excesso relativo d. O processo de verificação de chaves é similar ao descrito anteriormente, com a diferença de que se em algum momento dessa varredura encontrarmos o intervalo que contém o excesso relativo desejado, atualizamos v (se v não for um nó folha) para o filho correspondente à chave que contém a resposta, e interropemos o processo de verificação dos nós irmãos, retornando à *fwdSearch* a chave que contém a resposta, com o nó v atualizado. Tendo achado a resposta, o processo de subida na árvore é interrompido, caso contrário v é atualizado para o índice do seu nó pai e o processo de verificação dos nós irmãos é repetido.

Após encontrarmos o nó v, que contém o excesso d, iniciamos um estreitamento do intervalo a ser inspecionado, através do processo de descida na rmM-tree. Esta etapa é muita similar as demais, pecorremos as chaves de v verificando se a asserção $dr + R[v].key[l].m \leq d \leq dr + R[v].key[l].M$ é válida, atualizando o valor de dr, sempre que d não estiver compreendido dentro do intervalo de uma das chaves de v, quando a asserção for válida, atualizamos o valor de v, para sua respectiva chave. Repetimos esse processo até que cheguemos à um nó folha, que é quando executamos novamente *fwdKey*, varrendo o nó folha desde a sua primeira chave. Os algoritmos 7 e 8 mostram como o processo de descida na árvore funcionam, bem como o funcionamento completo de *fwdSearch*.

Exampo 8: Dado um nó em BP, codificado em $i = 21$, encontrar o primeiro índice $j > i$, tal que $excess(j) - excess(i) = -1$.

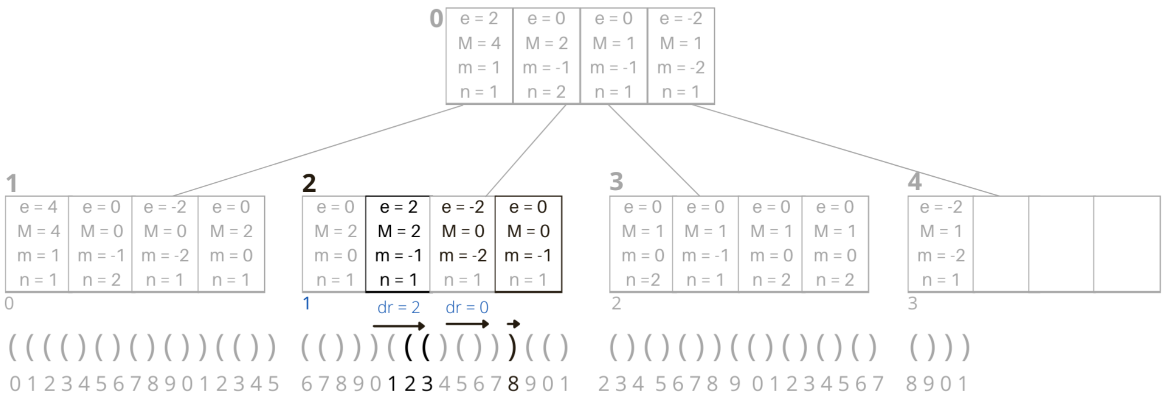
O processo para a obtenção dessa resposta é descrito a seguir:

O primeiro passo é identificar o índice do nó folha que cobre $i = 21 + 1$ (lembrando que a busca é por um $j > i$), neste caso $v = 3$. Precisamos também identificar a que chave de v, i pertence, temos que $\lfloor (21 - 16)/4 \rfloor = 1$. Iniciamos então a varredura, pecorrendo o nó v, a partir de $i + 1$ até o final da chave 1, ao chegar ao final da chave, temos como excesso computado até o momento (dr) 2, e portanto não encontramos o excesso buscado ($d = -1$).

Como ainda existem chaves em v que não foram inspecionados, pecorremos as chaves restantes, verificando a cada momento se $dr + R[3].keys[l].m \leq d \leq dr + R[3].keys[l].M$ é válida, passamos pela chave 2 e não encontramos o excesso buscado, atualizamos o valor de dr somando à ele $R[3].keys[2].e$, e inspecionamos a chave 3, onde o excesso $dr = d$ finalmente é alcançado. Interrompemos a análise dos nós da rmM-tree, e iniciamos uma análise mais detalhada, bit-a-bit da chave 3, a fim de encontrar a posição exata onde d ocorre. Durante essa inspeção, atualizamos o valor de dr em 1 ao inspecionarmos um bit 1, e em -1 ao visitarmos um bit 0. A varredura bit-a-bit, acontece a partir do ínicio da chave 3, e já no primeiro índice desta chave temos que $dr = d$, nesse momento significa que encontramos a resposta buscada, sendo que ela $j = 28$.

Perceba que aqui foi necessário a inspeção de 2 nós (excluindo o nó inicial), sem a necessida de subir na rmM-tree, ao passo que para o mesmo exemplo, na estrutura binária é necessário a inspeção de 5 nós (excluindo o nó inicial), subir um nível, e depois descer um nível.

Figura 3.2: Simulação da operação *fwdSearch(21,-1)*.



3.3.2 BwdSearch

Como dito anteriormente, o processo para calcular bwdSearch é completamente análogo à fwdSearch, devendo termos cuidado apenas com as questões relacionadas a simetria dos dados abordadas no capítulo 2. Tendo isso em mente podemos construir uma função que verifica se o excesso buscado está entre as chaves das folhas analisadas, assim como uma segunda função para inspecionarmos as chaves dos p vizinhos de um nó v visitado durante o processo de subida e descida na árvore. Os algoritmos 9, 10 e 11 mostram o pseudocódigo para as adaptações da estrutura multi-ária.

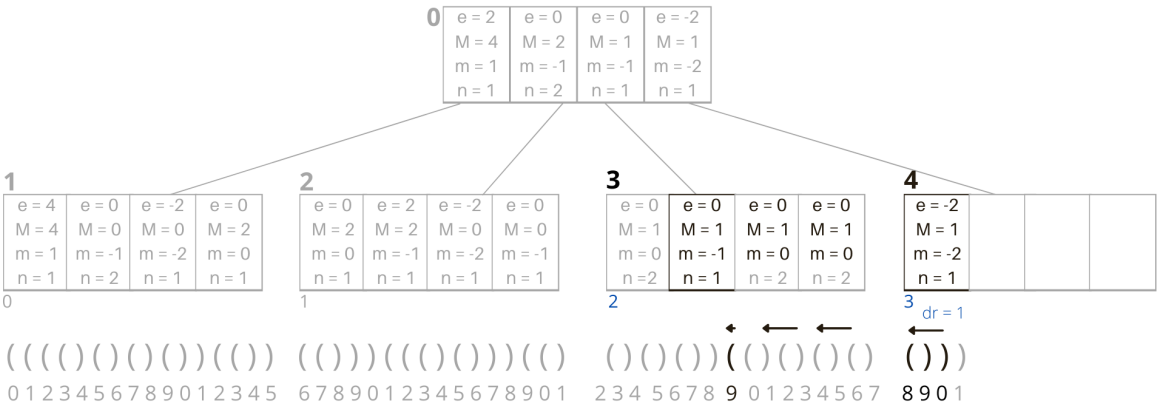
Apenas para reafirmar, temos as seguintes observações em relação à assimetria dos dados

em relação a bwdSearch[2]:

- A posição i é incluída na contagem de excessos;
- Adicionamos 1 à dr quando passamos por um bit 0, e subtraímos 1 de dr quando passamos por um bit 1;
- $excess(i)+d$ é alcançado quando $dr-R[v].e+R[v].m \leq d \leq dr-R[v].e+R[v].M$ for verdadeiro.

Exampo 9: Dado um nó em BP, codificado em $i = 50$, encontrar o índice $j < i$ mais à direita de i,tal que $excess(j) - excess(i) = 0$.

Figura 3.3: Simulação da operação *bwdSearch(50,0)*.



Ao identificarmos o nó a qual pertence o índice $i = 50$, temos que $v = 4$, temos também que a chave onde se localiza esse índice é a primeira deste nó, ou seja 0 ($\lfloor (50-48)/4 \rfloor$).

Inspecionamos a chave deste nó partindo de $i = 50$, até chegar ao início da mesma que é $s = 48$, ao terminarmos esta inspeção temos que $dr = 1$, como ternminarmos de inspecionar as chaves deste nó, verificamos quantos irmãos v possuí a sua esquerda, afim de inspecionarmos os mesmos. Atualizamos v para o seu antecessor, que passa a valer 3, pecorremos então cada chave de v buscando a chave onde a asserção $(dr-R[3].keys[l].e+R[3]].keys[l].m \leq d \leq dr-R[3].keys[l].e+R[3].keys[l].M)$ é válida. Ao verificarmos a chave de número 3 do nó 3, percebemos que a asserção definida não é válida, seguimos para a chave que precede a atual (2), e temos que d não está incluso nos intervalos de excesso mínimo e máximo da chave 2, atualizamos então a chave corrente para 1, e a asserção é validada. Como encontramos a chave que contém d, interrompemos a verificação dos nós da rmM-tree, e iniciamos uma

[2]Para mais detalhes, verificar seção 2.5.4

varredura bit a bit, do intervalo desta chave. Ao realizarmos a inspeção da chave 1, do nó 3, encontramos $dr = d$ em $j = 49$.

Para este exemplo fizemos a inspeção dos campos de 3 nós (excluindo o nó inicial), não sendo necessário subir na rmM-tree, já para a estrutura binária, como visto anteriormente, foi necessário, realizar a inspeção de 6 nós (excluindo o nó inicial), subindo 1 nível da estrutura, e depois descendo 2 níveis.

3.3.3 Derivadas

Não entraremos nos detalhes das operações derivadas de *fwdSearch* e *bwdSearch*, tendo em vista que as mesmas já foram abordadas anteriormente. Ademais, a tabela 3.1 lista as operações suportadas pela nossa implementação da rmM-tree k-ária até o momento de escrita deste trabalho, observe que ela não traz operações como a de menor ancestral comum (lca), e obtenção de filho mais profundo de um nó ($deepestNode$), isso ocorre porque as operações que são bases para estas ainda não foram implementadas, entretanto, destaca-se que o processo de adaptação destas operações base é muito similar ao que já foi exposto até aqui.

Algoritmo 5: Construção da range min-Max tree k-ária

Input: $BP[0,n-1]$, tabela C, tamanho de bloco b e de sub-bloco w (com b mod $w = 0$), número de folhas r, quantidade de nós $nNodes$

```
1  // Construção dos nós folhas
2  numKey ← 0
3  for k ← 0 to r − 1 do
4      v ← leafInTree(k); key ← 0
5      while R[v].nKeys < m and nunKey ≤ ⌈BP.size()/b⌉ do
6          R[v].keys[key].e ← 0; R[v].keys[key].m ← −w;
7          R[v].keys[key].M ← w; R[v].keys[key].n ← 0;
8          for p ← (numKey ∗ (b/w)) + 1 to ((numKey + 1) ∗ b)/w do
9              x ← bistread((p − 1) ∗ w)
10             if R[v].keys[key].e + C[x].M > R[v].keys[key].M then
                   R[v].keys[key].M ← R[v].keys[key].e + C[x].M;
11             if R[v].keys[key].e + C[x].m < R[v].keys[key].m then
                   R[v].keys[key].m ← R[v].keys[key].e + C[x].m;
12             else if R[v].keys[key].e + C[x].m = R[v].keys[key].m then
13                 R[v].keys[key].n ← R[v].keys[key].n + C[x].n
14             end
15             R[v].keys[key].e ← R[v].keys[key].e + C[x].e
16         end
17         R[v].nKeys ← R[v].nKeys + 1
18         numKey ← numKey + 1
19         key ← key + 1
20     end
21 end
22 // Construção dos nós internos e raíz
23 for v ← nNodes − r − 1 to 0 do
24     for key ← 0 to m − 1 and (v ∗ m) + key + 1 to nNodes − 1 do
25         child ← (m ∗ v) + 1 + key
26         R[v].keys[key].e ← R[child].keys[0].e
27         R[v].keys[child].m ← R[child].keys[0].m
28         R[v].keys[child].M ← R[child].keys[0].M
29         R[v].keys[child].n ← R[child].keys[0].n
30         for i ← 1 to R[child].nKeys − 1 do
31             R[v].keys[key].M =
                   max(R[child].keys[i].M, R[v].keys[i − 1].e + R[child].keys[i].M)
32             R[v].keys[key].m =
                   max(R[child].keys[i].m, R[v].keys[i − 1].e + R[child].keys[i].m)
33             if R[child].keys[i].m < R[v].keys[i − 1].e + R[child].keys[i].m then
34                 R[v].keys[key].n ← R[child].keys[i].n
35             end
36             else if R[child].keys[i].m = R[v].keys[i − 1].e + R[child].keys[i].m then
                   R[v].keys[key].n ← R[child].keys[i].n + R[child].keys[i − 1].n ;
37             R[v].keys[key].e ← R[child].keys[i].e
38         end
39     end
40 end
```

Algoritmo 6: Verificando as chaves de um nó folha através de $fwdKey(i, v, key, k, d, \&dr)$

Input: Nó v, e folha correspondente k, posição a partir do qual devemos iniciar a busca (índice i e key), excesso relativo buscado, e excesso relativo computado em cada chave inspecionada (d e $\&dr$).

Output: Posição j ou $BP.size()$ caso d não seja encontrado.

```
1  for key to R[v].nKeys − 1 do
2      if (dr + R[v].keys[key].m ≤ d ≤ dr + R[v].keys[key].M) then
3          j ← fwdBlock(i, d, dr) // Algoritmo 3
4          if dr = d then
5              return j
6          end
7      end
8      dr ← dr + R[v].keys[key].e
9      i ← (m ∗ k + key + 1) ∗ b − 1 // calcula o fim da chave atual
10     if dr = d then
11         return i
12     end
13 end
14 return BP.size()
```

Algoritmo 7: Busca o excesso relativo nos irmãos de v através de $fwdVerifySibling(\&v, \&dr, d)$

Input: Nó $\&v$, excesso relativo buscado e excesso relativo computado em cada chave inspecionada (d e $\&dr$).

Output: key ou $BP.size()$ caso o intervalo que contém d não seja encontrado.

```
1  parent ← ⌊(v − 1)/m⌋
2  n_sibling ← v − (parent ∗ m)
3  v ← v + 1
4  while n_sibling to R[parent].nKeys − 1 and v to num_nodes − 1 do
5      for key ← 0 to R[v].nKeys − 1 do
6          if (dr + R[v].keys[key].m ≤ d ≤ dr + R[v].keys[key].M) then
7              if v ≤ numberNodes − numberLeaves then v ← (v ∗ m) + 1 + key;
8              return key
9          end
10         dr ← dr + R[v].keys[key].e
11         if dr = d then
12             if v ≤ numberNodes − numberLeaves then v ← (v ∗ m) + key + 2;
13             return key + 1
14         end
15     end
16     n_sibling ← n_sibling + 1
17     v ← v + 1
18 end
19 return BP.size()
```

Algoritmo 8: Busca por excesso no intervalo $[i+1, BP.size()-1]$ através de $fwdSearch(i,d)$

Input: Índice i a partir do qual a busca deve ser feita e excesso relativo buscado d.
Output: Posição j onde ocorre o excesso d ou $BP.size()$ caso d não exista no intervalo definido.

```
1  dr ← 0
2  k ← ⌊(i+1)/(b*m)⌋
3  v ← leafInTree(k)
4  key ← ⌊((i+1) − (k*b*m))/b⌋
5  j ← fwdBlock(i,d,dr)// Algoritmo 3
6  if dr = d then return j;
7  key ← key+1
8  if key < R[v].nKeys then
9  |   j ← fwdKey((m*k+key)*b−1,v,key,k,d,dr)
10 |   if dr = d then return j;
11 end

12 // Inicia o processo de subida na rmM-tree
13 while v ≠ 0 and fwdVerifySibling(v,dr,d) = BP.size() do
14 |   v ← ⌊(v−1)/m⌋
15 end
16 // Chegamos ao nó raíz, e d não está em nenhum dos seus
       filhos
17 if v = 0 and key = v.size() then return v.size();

18 // Inicia descida em árvore
19 while v ≤ numberNodes − numberLeaves do
20 |   for ke ← 0 to v.nKeys − 1 do
21 |   |   if (dr+v[chave].m ≤ d ≤ dr+v[chave].M) then
22 |   |   |   v ← (v*m)+1+chave
23 |   |   |   key ← 0
24 |   |   |   break
25 |   |   end
26 |   |   else dr ← dr+v[chave].e ;
27 |   end
28 end
29 k ← numLeaf(v)
30 i ← (m*k*chave)*sizeBlock // Calcula o índice inicial da
       folha que queremos inspecionar
31 j ← fwdKey(i−1,v,0,k,d,dr)
32 if dr = d then return j;
33 return BP.size()
```

Algoritmo 9: Verificando as chaves de um nó folha através de $bwdKey(i, v, key, k, d, \&dr)$

Input: Nó v e folha correspondente k, posição a partir do qual devemos iniciar a busca (índice i e key), excesso relativo buscado e excesso relativo computado em cada chave inspecionada (d e $\&dr$).

Output: Posição j ou -1 caso d não seja encontrado.

```
1  for key downto 0 do
2      if (dr − R[v].keys[key].e + R[v].keys[key].m ≤ d ≤
         dr − R[v].keys[key].e + R[v].keys[key].M) then
3          j ← bwdBlock(i, d, &dr) // Análogo ao algoritmo 3
4          if dr = d then return j;
5      end
6      dr ← dr − R[v].keys[key].e
7      i ← (m ∗ k + key) ∗ b − 1 // calcula o início da chave atual
8      if dr = d then return i;
9  end
10 return −1
```

Algoritmo 10: Busca o excesso relativo nos irmãos de v através de $bwdVerifySibling(\&v, \&dr, d)$

Input: Nó $\&v$, excesso relativo buscado e excesso relativo computado em cada chave inspecionada (d e $\&dr$).

Output: key ou -1 caso o intervalo que contém d não seja encontrado.

```
1  parent ← ⌊(v − 1)/m⌋
2  n_sibling ← v − (parent ∗ m) − 1
3  v ← v − 1
4  while n_sibling downto 0 and v downto 0 do
5      for key ← R[v].nKeys − 1 to 0 do
6          if (dr − R[v].keys[key].e + R[v].keys[key].m ≤ d) and
             (d ≤ dr − R[v].keys[key].e + R[v].keys[key].M) then
7              if v ≤ numberNodes − numberLeaves then v ← (v ∗ m) + 1 + key;
8              return key
9          end
10         dr ← dr − R[v].keys[key].e
11         if dr = d then if v ≤ numberNodes − numberLeaves then
             v ← (v ∗ m) + key;
12         return key ;
13     end
14     if n_sibling − 1 > 0 then v ← v − 1;
15     n_sibling ← n_sibling − 1
16 end
17 return −1
```

Algoritmo 11: Busca por excesso no intervalo $[i+1, BP.size()-1]$ através de $bwdSearch(i,d)$

Input: Índice i a partir do qual a busca deve ser feita e excesso relativo buscado d.
Output: Posição j onde ocorre o excesso d ou -1 caso d não exista no intervalo definido.

```
1  dr ← 0
2  k ← ⌊i/(b∗m)⌋
3  v ← leafInTree(k)
4  key ← ⌊((i+1)−(k∗b∗m))/b⌋
5  j ← bwdBlock(i,d,&dr)// Análogo ao algoritmo 3
6  if dr = d then return j;
7  key ← key−1
8  if key ≥ 0 then
9  |  j ← bwdKey((m∗k+key+1)∗b−1,v,key,k,d,&dr)
10 |  if dr = d then return j;
11 end

12 // Inicia o processo de subida na rmM-tree
13 while v ≠ 0 and bwdVerifySibling(&v,&dr,d) = −1 do
14 |  v ← ⌊(v−1)/m⌋
15 end
16 // Chegamos ao nó raíz, mas d não está em nenhum dos
       seus filhos
17 if v = 0 and key = −1 then
18 |  return −1
19 end

20 // Desce nível a nível da árvore, até que v corresponda
       ao índice de um dos nós folhas
21 while v ≤ numberNodes − numberLeaves do
22 |  for key ← R[v].nKeys−1 to 0 do
23 |  |  if (dr−R[v].keys[key].e+R[v].keys[key].m ≤ d ≤
   |  |   dr−R[v].keys[key].e+R[v].keys[key].M) then
24 |  |  |  v ← (v∗m)+1+chave
25 |  |  |  key ← R[v].nKeys−1
26 |  |  |  break
27 |  |  end
28 |  |  else dr ← dr−R[v].keys[key].e ;
29 |  end
30 end

31 k ← numLeaf(v)
32 if d = dr then return (k∗m∗b)+(key∗b)−1;
33 j ← bwdKey((k∗m∗b)+((key+1)∗b)−1,v,key,k,d,&dr) if dr = d then
     return j;
34 else return −1 ;
```

Tabela 3.1: Operações suportadas pela rmM-tree k-ária

Operação	**Retorno**
fwdSearch(i,d)	Índice j, tal que $excess(j) = excess(i) + d$
bwdSearch(i,d)	Índice j, tal que $excess(j+1,i) = -d$
findClose(i) / findOpen(i)	Posição do parênteses corresponde à $BV[i]$
enclose(i)	Posição do parênteses de abertura que envolve $BV[i]$
$rank_1$(i) / $rank_0$(i)	Número de parênteses abrindo/fechando em $BV[0,i]$
$select_1$(i) / $select_0$(i)	Posição do i-ésimo parênteses de abertura/fechamento
preRank(i) / postRank(i)	rank de *i* calculado a partir de um percurso *preorder ou postorder*
preSelect(i) / postSelect(i)	retorna o nó com *preorder/postorder i*
isLeaf(i)	Verifica se $BV[i]$ codifica uma folha
isAncestor(i,j)	Verifica se o nó codificado em *i* é ancestral de j
depth(i)	Profundidade do nó *i*
parent(i)	Obtém o pai do nó *i*
firstChild(i) / lastChild(i)	Retorna o primeiro/último filho do nó codficiado em $BV[i]$
nextSibling(i) / prevSibling(i)	Primeiro irmão à direita/esquerda de *i*
subtreeSize(i)	Número de nós enraízados na subárvore de *i*
levelAncestor(i,d)	Ancestral j de *i* tal que $depth(j) = depth(i) - d$
levelNext(i) / levelPrev(i)	Nó à direita/esquerda de *i* com a mesma profundidade de *i*.
levelLeftMost(d) / levelRightMost(d)	Nó mais à esquerda/direita, com profundidade d.
leafRank(i)	Número de folhas à esquerda da folha codificada em *i*
leafSelect(i)	$i-$ésima folha em $BV[0,size-1]$
leftMostLeaf(i)	folha codificada em j, mais à direita de *i*, tal que $j < i$
rightMostLeaf(i)	folha codificada em j, mais à direita possível, tal que $j \leq i$

4

Resultados Experimentais

Este capítulo discorre sobre os procedimentos realizados para a obtenção dos resultados comparativos da rmM-tree proposta por SADAKANE K. E NAVARRO (2010), com a rmM-tree k-ária proposta neste trabalho. O mesmo está dividido da seguinte forma: a seção 4.1 apresenta o conjunto de dados usado na avaliação experimental. As seções seguintes apresentam as configurações da máquina usada para realizar os testes (Seção 4.2), o modo como os mesmos foram realizados (Seção 4.3), e por últimos os resultados obtidos (Seção 4.4).

4.1 Base de dados

Afim de realizar testes que comprovem o desempenho das estruturas implementadas na ptrática, usamos em nossos testes quatro conjuntos de dados do mundo real, estes conjuntos estão disponíveis no endereço eletrônico www.inf.udec.cl. A tabela 4.1 mostra um resumo dos conjuntos usados, a primeira coluna da tabela refere-se ao nome do conjunto de dados, a segunda coluna indica o tamanho do conjunto de dados em MB, a coluna de número 3 mostra a quantidade de parênteses do respectivo conjunto de dados. A quarta coluna exibe a quantidade nós da árvore representada pela sequência de parêntesses fornecida, e por fim, a quinta coluna mostra o número de entradas usadas em nossos testes para cada conjunto de dados.

Tabela 4.1: Conjunto de dados usados nos testes experimetais

Conjunto de dados	Tamanho (MB)	Quantidade de parênteses	Tamanho da árvore representada	Número de entradas usadas
Complete tree (ctree)	18	2.147.483.644	1.073.741.822	1.000.000
DNA	135	1.154.482.174	577.241.087	100.000.000
Proteins (prot)	82	670.721.006	335.36203	3.000.000
Wikipedia (wiki)	13	498.753.914	249.376.957	1.000.000

Conforme nos mostra o site, o conjunto *Complete tree* represeta uma árvore binária completa de profundidade igual à 30, ao passo que a sequência de parênteses balanceados *DNA*/*Proteins* representam uma árvore de sufixos de DNA e proteínas, respectivamente. Por último o conjunto *Wikipedia* representa o XML extraído da Wikipédia no dia 12 de janeiro de 2015 (NAVARRO, ????).

4.2 Configuração

Para realizar os testes de validação e de comparação de desempenho foi utilizado o servidor Turing, disponível no IFB. A máquina usada possuí as seguintes configurações:

- Arquitetura: x86
- Processador: Intel Xeon Gold 5120
- Frequência base: 2,20GHz
- Frequência máxima: 3,20 GHz
- Threads por core: 2
- Cores: 28
- Cache L1: 896 KiB
- Cache L2: 28 MiB
- Cache L3: 38.5 MiB
- Memória RAM total: 527,03 Gb

4.3 Experimentos

Os conjuntos mostrados acima, fornecem árvores gerais no formato de parênteses balanceados, de modo que foi preciso pré-processar os conjuntos antes da execução dos testes, afim de adaptar os mesmos à estrutura de vetores de bits, que serviu como base para a construção das Range min-Max trees,

Para realizarmos uma comparação justa entre a estrutura proposta por SADAKANE K. E NAVARRO (2010) e a estrutura proposta neste trabalho, construímos a rmM-tree no seu formato clássico, e a rmM-tree multi-ária, ambas disponíveis no repositório github.com/DanyelleAngelo/TCC, para tanto foi usada a linguagem de programação C++.

Objetivando garantir a asserção das respostas retornadas por ambas estruturas, fizemos testes unitários usando o framework Google Tests. Incialmente foram criados testes com árvores gerais pequenas, comparamos os resultados da nossa implementação da rmM-tree binária, com os resultados da rmM-tree implementada na biblioteca *Succint Data Structure Library* (SDSL). Para algumas operações foi necessário computar o valor esperado usando uma série de operações básicas de vetores de bits, tendo em vista que a SDSL não fornecia todas as operações necessárias para os testes da nossa estrutura.

Após os primeiros testes, com conjuntos de dados pequenos, expandimos o nosso escopo de testes, aumentando o tamanho da árvore de entrada usada, para esses testes usamos o

conjunto de dados *Wikipedia*, geramos então diversos vetores, com valores pseudo-aleatórios, correspondentes aos limites do conjunto usado. Cada um desses vetores foi gerado com uma semente diferente, para que pudéssemos ter diferentes parâmetros de pesquisa. Para o caso das operações que realizam o percurso em árvore somente mediante à um parâmetro de entrada correspondete à um "("ou de ")"criamos vetores específicos, que continham os índices do conjunto de dados correspondente ao elemento desejado. Foram geradas sequências de em média 25.000.000 entradas para todas as funções disponíveis na nossa implementação.

Após realizar os testes de validação com a range-min-Max tree binária, iniciamos a construção da Range min-Max tree k-ária, seguindo a mesma metodologia, realizando inicialmente testes de validação com árvores gerais pequenas, e depois gerando testes com árvores gerais do mundo real.

Tendo concluído os testes de validação, iniciamos os testes de desempenho, para tanto foi usado o framework Google Benchmark, os testes nessa etapa caracterizaram-se principalmente pela refataração do código das rmM-trees construídas, na tentativa de melhorar o desempenho revelado pelo framwork. A cada refatoração, invocamos os testes unitários e os testes desempenho, afim de validarmos o valor das alterações. Em um primeiro momento, para a estrutura da rmM-tree k-ária, visando evitar o pior caso (quando o nó que estamos visitando é um dos primeiros descendentes de seu pai e precisamos visitar todos os seus irmãos, e ao final não encontramos a resposta), construímos uma versão alternativa dessa estrutura. Nessa versão alternativa, excluímos uma otimização de percurso pela rmM-tree proposta por NAVARRO (2016), na otimização proposta pelo autor, antes de visitarmos o pai de um nó, visitamos o irmão mais à direita daquele nó, e caso a resposta não esteja no irmão deste nó, atualizamos o mesmo para o índice do seu nó pai, que é exatamente no que consiste o nosso pior caso. Assim, a versão alternativa da rmM-tree k-ária, pula esta etapa de verificação dos irmãos de um nó v, analisando imediamente os valores das chaves do pai de v ao constatar que a resposta não se encontra em v.

Essa última versão da rmM-tree apresentou uma melhoria de desempenho, mas mesmo após essa versão os resultados, ainda não eram os esperados, fizemos então uma segunda otimização, desta vez, em ambas as árvores (para manter as mesmas condições de testes). Esta alteração visava melhorar diretamente a forma como um intervalo de bits é lido durante o processo de inspeção de bloco, esse processo envolve ler um intervalo de bits s, e , convertê-lo em número decimal, e então fazer uma pesquisa em uma tabela de excessos pré-computada (que acelera a checagem de campos da rmM-tree). O problema é que inicialmente esse processo conversão era feito iterativamente, o que acrecescentava tempo condiserável em nossos testes, alteramos esse processo iterativo de conversão, para um processo bit-a-bit, usando operações de deslocamento de bits e uma tabela de revesrsão (para que a conversão pudesse ser feita do bit mais significativo para o menos significativo). Realizamos novamente os testes unitários para ambas as estruturas (incluindo as duas versões da rmM-tree k-ária), após termos certeza de que a alteração feita, não impacatava negativamente nos valores retornados pelas nossas funções, realizamos os testes de benchmark para as estruturas implementadas. Notamos uma

melhoria significativa para as nossas estruturas, e neste momento ao comparar a primeira versão da rmM-tree k-ária, com a segunda versão, notamos que o desempenho da primeira versão era superior se comparado ao desempenho da segunda, isso porque, apesar do pior caso exposto acima, a verificação premeditada dos irmãos de um nó (presente na primeira versão da rmM-tree), evita a troca desncessária de dados entre cache e memória RAM, o que vai de encontro ao cerne deste trabalho.

Tendo feito estas otimizações e definindo a versão da rmM-tree k-ária a ser usada (primeira versão), aumentamos a abrangência dos nossos de testes de benchmark, usando os demais conjuntos de dados expostos na tabela 4.1, neste momento aumentamentamos também o número de entradas, que nos primeiros testes de benchmark eram cerca de 100.000, para os valores demonstrados nessa mesma tabela. A próxima seção, expõe os resultados de benchmark obtidos após os testes definitivos.

4.4 Resultados

O tempo gasto por cada operação da rmM-tree binária e k-ária, foi exportado para um arquivo *.csv*, e a partir destes resultados, foi gerado através da linguagem de programação Python, os gráficos de tempo médio gasto por cada operação.

Os gráficos de barras aninhadas, agrupam os resultados para cada tipo de árvore por operação. As árvores geradas para os testes são rmM-tree binária (em azul), rmM-tree 4-ária (barra laranja), rmM-tree 8-ária (verde), e rmM-tree 16-ária (vermelho), o tamanho de bloco usado em todos os testes foi definido como 32 (ou seja, cada intervalo coberto tem tamanho 32), e a constante w que divide b, foi setada como 16, mantendo assim a tabela C, que auxilia na montagem da árvore e na verificação dos dados em cache.

O eixo y dos gráficos se refere ao tempo médio para l pesquisas, no eixo x podemos ver os tipos de árvores comparadas bem como as operações realizadas. Ressalta-se novamente que as operações *minExcess, minSelectExcess, minCount* e suas derivadas não foram implementadas na nossa versão k-ária da rmM-tree, e por isso o desempenho das mesmas não foram levados em consideração pela nossa análise, além destas também não realizamos testes de benchmark para as operações que derivam exclusivamente das operações suportadas pela estrutura de vetores de bits.

Ao analisar os gráficos, perceba que de modo geral, para estrutura k-ária, a medida que aumentamos a ordem da árvore, diminuímos também o tempo médio das operações. Entretanto o mesmo não é válido, quando olhamos para o tempo médio gasto em uma operação da estrutura binária em comparação com a estrutura 16-ária, acredita-se que isso se deve à quantidade de otimizações a nível de código existente na estrutura binária, frente a quantidade existente na estrutura k-ária; na estrutura binária, por exemplo, o percurso de subida/descida na rmM-tree, é interrompido sempre que alcançamos o último nó de um nível, em uma pesquisa por um índice $j > i$, ou quando alcançamos o primeiro nó de um nó nível, em uma busca por um índice $j < i$,

onde *i* é o parâmetro de entrada.

Destaca-se por último que a diferença de desempenho relativamente grande entre as estruturas analisadas para fwdSearch e bwdSearch, pode se dever a casos em que o excesso *d* buscado não existe para determinado nó codificado em *i*, tendo em vista que os valores de excesso *d* e os nós analisados são aleatórios, ou seja, existem casos em que não existem outros nós *d* níveis acima/abaixo de um nó codificado em *i*, este refere-se exatamente ao que expomos no paragráfo anterior, entretanto a quantidade de *respostas não encontradas* para operações de *fwdSearch/bwdSearch* tendem a ser maiores, haja vista que para operações como *findClose/findOpen* sempre haverá uma resposta, desde que os filtros para os vetores que servem como parâmetros de entrada, sejam configurados corretamente.

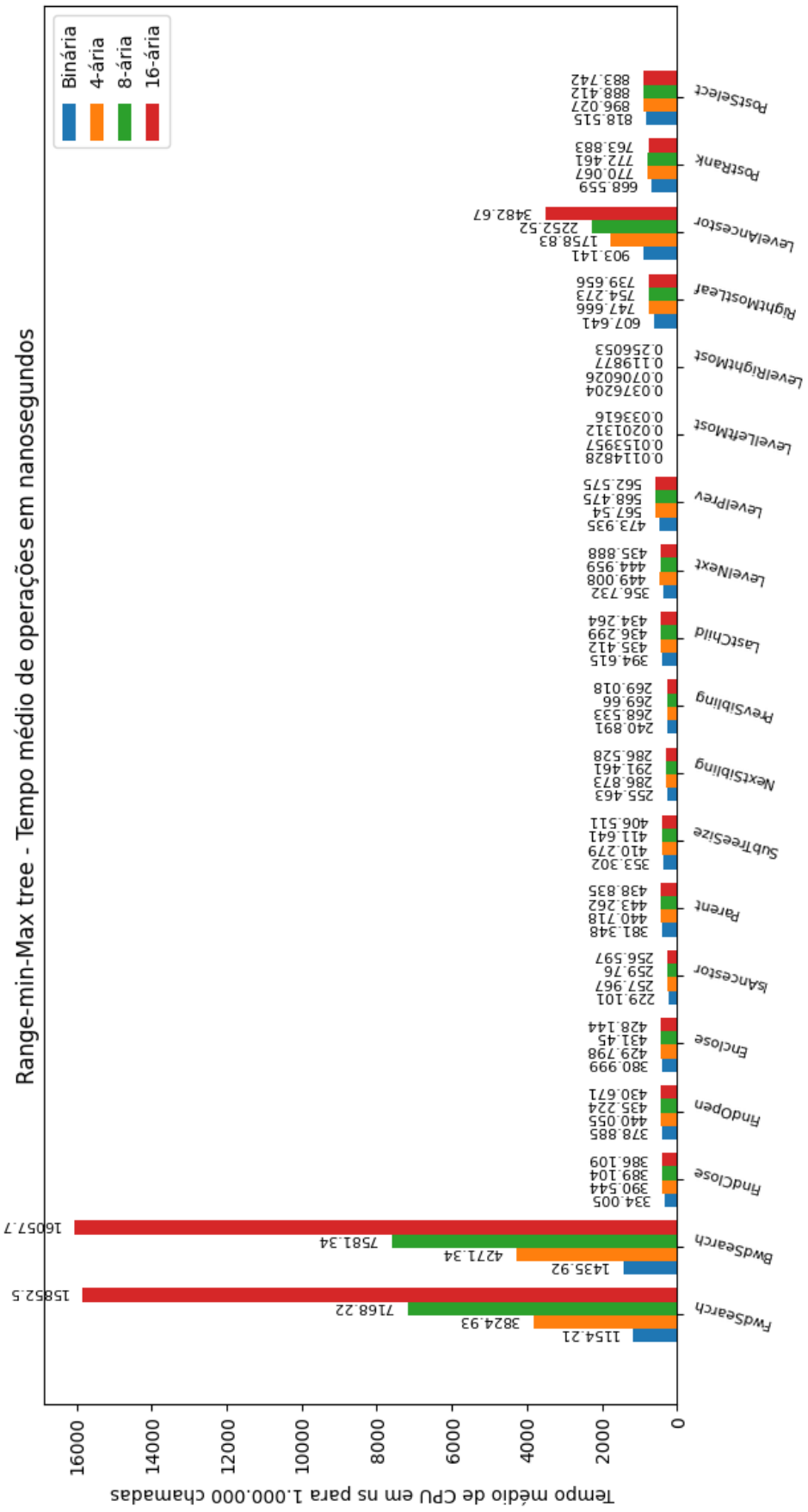


Figura 4.1: [
Tempo médio de operações sobre o conjunto Complete Tree]

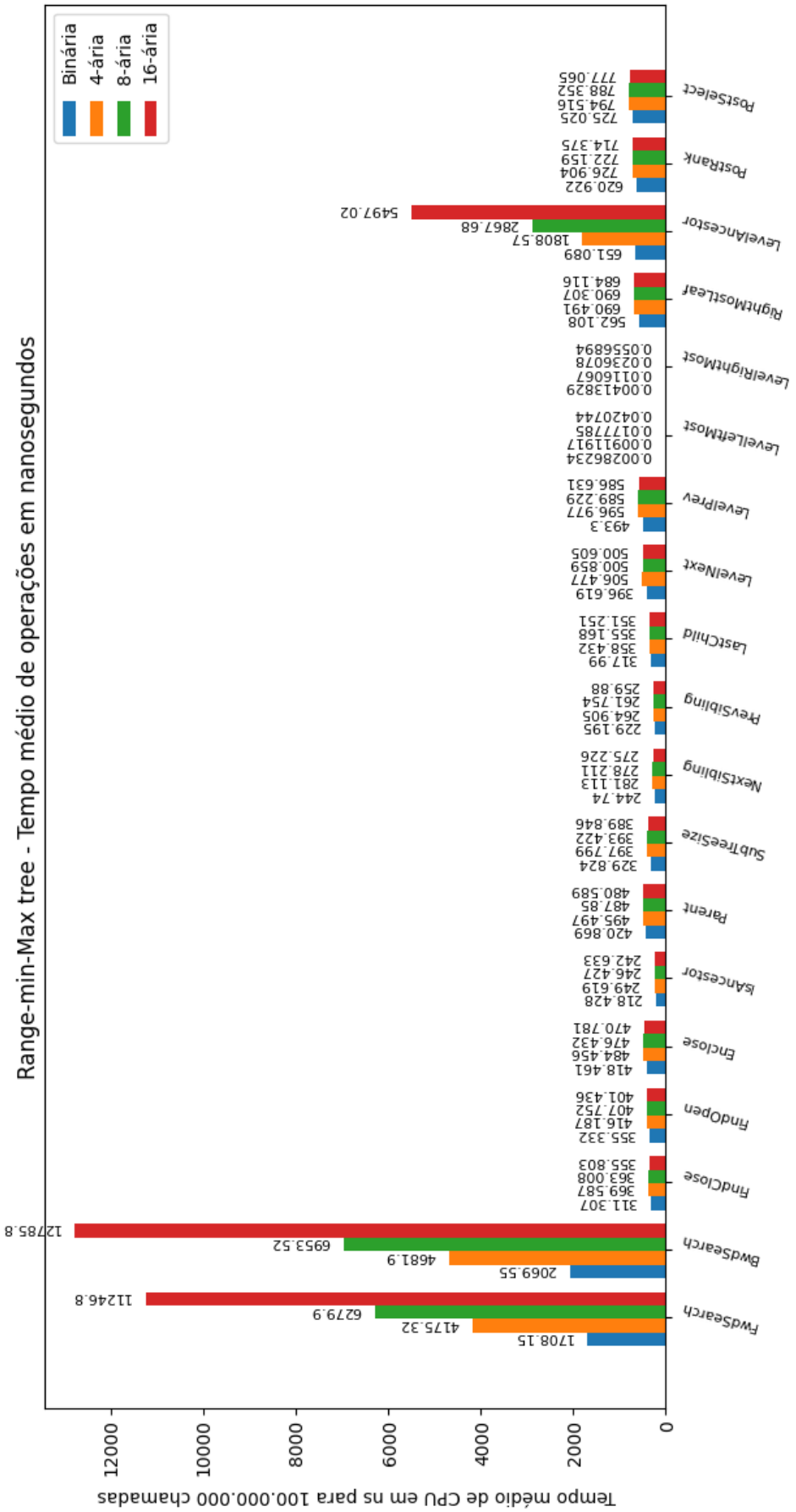


Figura 4.2: [
Tempo médio de operações sobre o conjunto DNA]

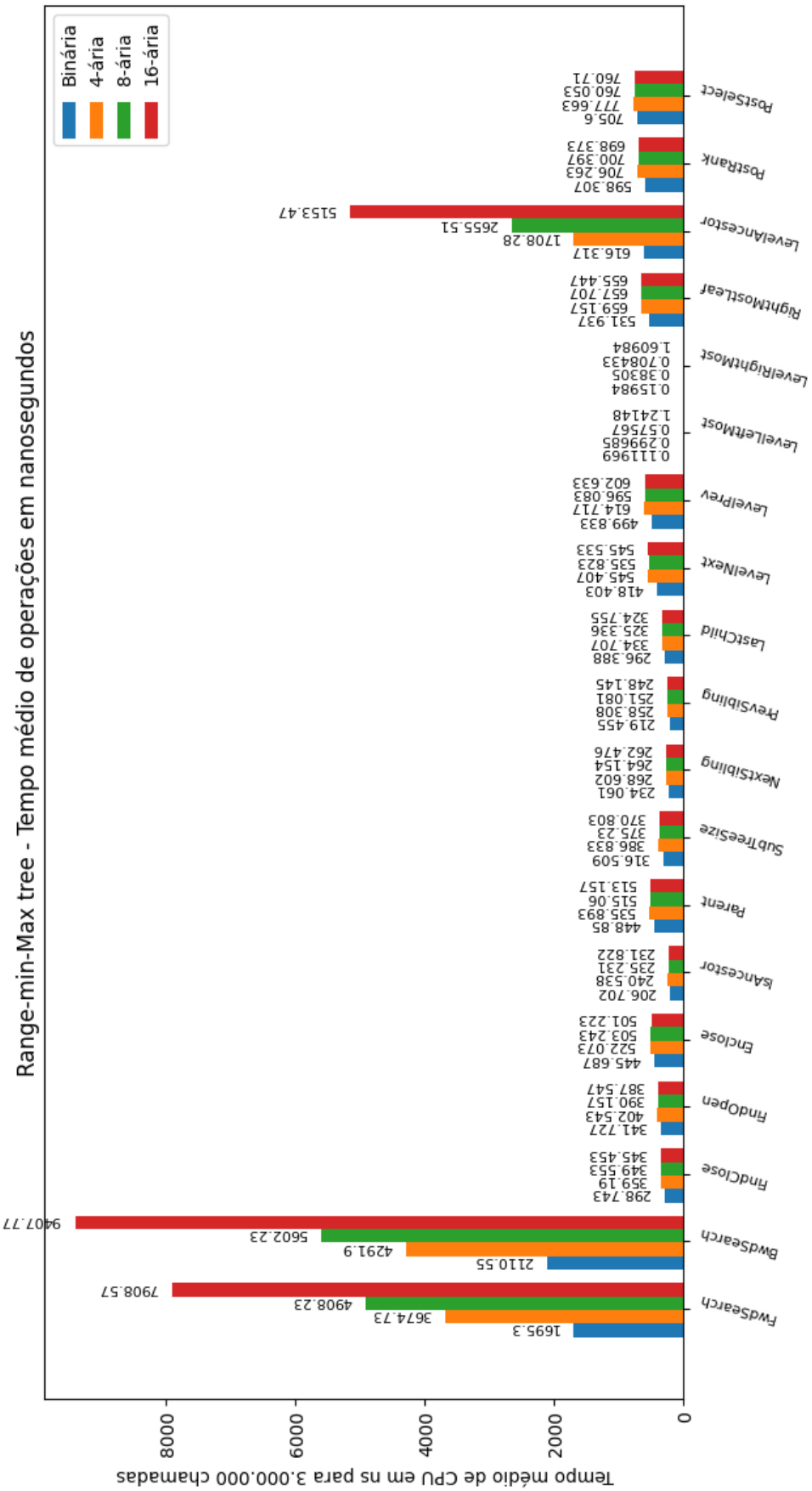


Figura 4.3: [
Tempo médio de operações sobre o conjunto Proteins]

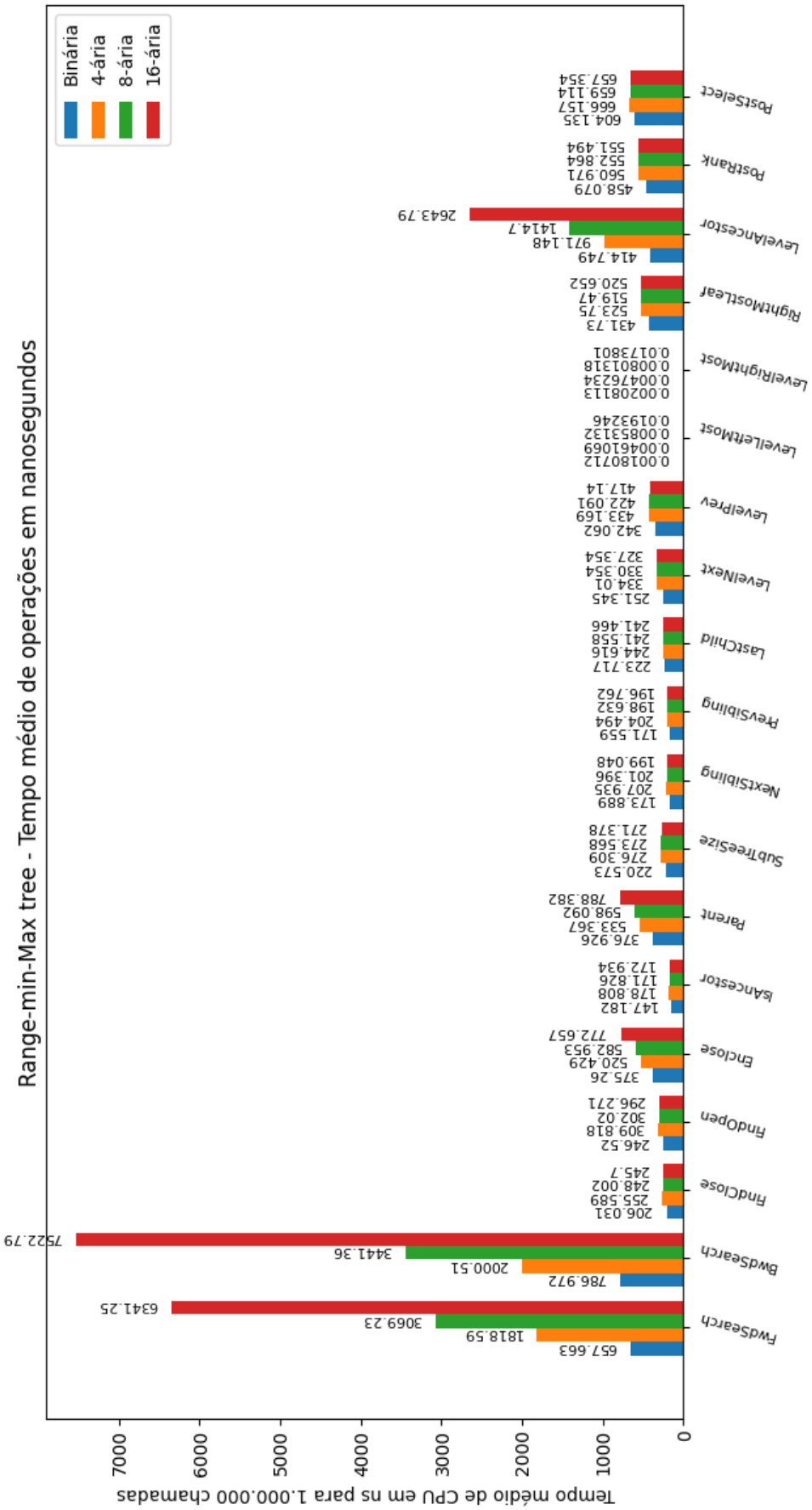


Figura 4.4: [
Tempo médio de operações sobre o conjunto Wikipedia]

5

Considerações Finais

Com o surgimento de novos dispositivos e o aumento na produção de dados, temos um grande desafio a superar, frente ao gargalo de comunicação existente entre memória e processador. Soluções como hierarquia de memória objetivam diminuir esse gargalo, entretanto quando se trabalha com uma alta quantidade de dados, os mesmos tendem a ficar distribuídos entre os diferentes níveis de memória. Com isso o processador frequentemente precisa realizar buscas em memórias nos níveis mais inferiores da hierarquia, o que acarreta em um acréscimo de tempo considerável no tempo de resposta, devido a baixa latência dessas memórias.

As estruturas de dados sucintas, objetivam representar e operar sobre os dados usando um espaço reduzido, o que gera um aproveitamento melhor da hierarquia de memória. Assim um dos objetivos deste trabalho era compreender e enfatizar a importância do estudo dessas estruturas. Como objetivo central, tínhamos a construção de uma versão alternativa de uma estrutura sucinta denominada range min-Max tree, proposta por SADAKANE K. E NAVARRO, essa estrutura permite a navegação de forma eficiente em árvore, através de valores de excesso máximos e mínimos em intervalos. Entretanto a mesma é construída na forma de árvore binária, uma estrutura que possuí baixo fator de ramificação, nosso objetivo portanto era aumentar o fator de ramificação dessa estrutura, visando ter uma diminuição do número de faltas de cache, como visto em (RAO J. E ROSS, 2000), o que reduziria significativamente o tempo gasto para realizar operações diversas.

Construímos e analisamos, para tanto, uma versão da rmM-tree de NAVARRO, e 3 versões de uma rmM-tree k-kária, estas três, compreendem uma rmM-tree 4-ária, 8-ária, e uma rmM-tree 16-ária. De modo geral, os resultados obtidos a partir destas implementações não foram satisfatórios, em nossos testes observamos um desempenho médio melhor em relação a rmM-tree k-ária, para esta estrutura, à medida que a ordem da árvore gerada aumenta, o tempo médio das operações é reduzido, mostrando o impacto que a ordem da árvore tem no resultado final. Entretanto, o mesmo não pôde ser observado para a rmM-tree binária em comparação a multi-ária, isso se deve ao fato das técnicas de código usadas para a implementação da rmM-tree binária serem superiores as técnicas usadas para a construção da rmM-tree k-ária.

Para trabalhos futuros temos como objetivo reduzir o tempo das operações, através da otimização da implementação da rmM-tree k-ária. Buscaremos também melhorar o escopo dos

nossos testes afim de monitorar de modo mais claro e eficaz o uso da cache, o que envolve ampliar o escopo das ferramentas usadas. Sugere-se também investigar o impacto dessa proposta em diferentes ambientes, visando entender o efeito dessa estrutura diante de diferentes configurações de hardware, por último temos como objetivo a implementação das demais operações de percurso em árvores suportadas pela Range min-max tree clássica e que ainda não foram implementadas em sua versão k-ária.

Referências

ARROYUELO, D. t. Succinct Trees in Practice. In: MEETING ON ALGORITHM ENGINEERING & EXPERMIMENTS, Austin, Texas, USA. **Proceedings...** Society for Industrial and Applied Mathematics, 2010. p.84–97. (ALENEX '10).

CARVALHO, C. The gap between processor and memory speeds. In: IEEE INTERNATIONAL CONFERENCE ON CONTROL AND AUTOMATION. **Proceedings...** [S.l.: s.n.], 2002.

CISCO. **Cisco Annual Internet Report (2018–2023)**. [S.l.]: Cisco, 2020.

COIRA, F. S. Compact data structures for large and complex datasets. In: OF . **Anais...** [S.l.: s.n.], 2017.

CORDOVA J. E NAVARRO, G. Simple and Efficient Fully-Functional Succinct Trees. In: GBR. **Anais...** Theoretical Computer Science, 2016. v.656, n.PB, p.135–145.

CORMEN, T. H. t. **Algoritmos**. 3.ed. Rio de Janeiro, RJ, Brasil: Elsevier, 2012.

DOMO. **Data Never Sleeps 8.0**. [S.l.]: Domo, 2020.

EFNUSHEVA D.; CHOLAKOSKA, A. e. T. A. M. A survey of different approaches for overcoming the processor-memory bottleneck. **Computer Science and Information Technology**, [S.l.], v.9, n.2, p.151–163, 2017.

HANKINS R. A. E PATEL, J. M. Effect of Node Size on the Performance of Cache-Conscious B+-trees. In: THEORETICAL COMPUTER SCIENCE. **Anais...** ACM SIGMETRICS international conference, 2003. v.29, p.475–476.

HENNESSY J. L. E PATTERSON, D. A. **Arquitetura de Computadores**: uma abordagem quantitativa. Quinta.ed. Rio de Janeiro, RJ, BR: Elsevier, 2014.

MAHAPATRA N. R. E VENKATRAO, B. The Processor-Memory Bottleneck: problems and solutions. **XRDS**, New York, NY, USA, v.5, n.3es, p.2–es, Apr. 1999.

MUNRO J. I. E RAMAN, V. Succinct representation of balanced parentheses, static trees and planar graphs. In: ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 38. **Proceedings...** Society for Industrial and Applied Mathematics, 1997. p.118–126.

NAVARRO, G. **Parênteses balanceados**.

NAVARRO, G. **Compact data structures**:a practical approach. 1.ed. New York, NY, USA: Sheridan Books, Inc, 2016.

RAO J. E ROSS, K. A. Making B+- Trees Cache Conscious in Main Memory. In: SOCIETY FOR INDUSTRIAL AND APPLIED MATHEMATICS, New York, NY, USA. **Anais...** Association for Computing Machinery, 2000.

REINSEL D. E GANTZ, J. e. R. J. **The Digitization of the World, From Edge to Core**. [S.l.]: International Data Corporation (IDC), 2018.

RUSSELL S. E NORVIG, P. **Inteligência Artificial**. 3.ed. Rio de Janeiro, RJ, Brasil: Elsevier, 2013.

SADAKANE K. E NAVARRO, G. Fully-Functional Succinct Trees. In: ASSOCIATION FOR COMPUTING MACHINERY. **Anais...** Society for Industrial and Applied Mathematics, 2010.