



**INSTITUTO  
FEDERAL**  
Brasília

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

**UMA PROPOSTA DE RANGE MIN-MAX TREE K-ÁRIA PARA CONSULTAS  
SOBRE ÁRVORES SUCINTAS**

**Por**

***DANYELLE DA SILVA OLIVEIRA ANGELO***

**Trabalho de Graduação**

BRASÍLIA/2021



Danyelle da Silva Oliveira Angelo

**UMA PROPOSTA DE RANGE MIN-MAX TREE K-ÁRIA 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 como requisito parcial para obtenção  
do grau de Bacharel em Ciência da Computação.*

**Orientador:** Daniel Saad Nogueira Nunes

BRASÍLIA  
2021

---

Danyelle da Silva Oliveira Angelo

Uma proposta de Range min-Max tree k-ária para consultas sobre árvores sucintas/  
Danyelle da Silva Oliveira Angelo. – BRASÍLIA, 2021-  
Orientador Daniel Saad Nogueira Nunes

Trabalho de Graduação – Instituto Federal de Educação, Ciência e Tecnologia de Brasília,  
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 Silva Oliveira Angelo** ao curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia de Brasília sob o título **Uma proposta de Range min-Max tree k-ária 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  
IFB, Taguatinga

---

Prof. Me. João Victor de Araujo Oliveira  
IFB, Taguatinga

---

Prof. Dr. Felipe Alves da Louza  
Faculdade de Engenharia Elétrica/UFU



## 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 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. Este trabalho se concentra na proposta de uma estrutura range min-Max tree k-ária, visando um maior fator de ramificação, e em decorrência disso uma minimização do número de eventuais faltas de cache (*cache misses*), contribuindo para a melhoria do desempenho das operações suportadas pela rmM-tree.

**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 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. This work focuses on the proposal of a min-Max tree k-ary range, aiming a higher branching factor, and as a result a minimization of the number of possible cache failures (*cache failures*), contributing to the improvement of the performance of the operations supported by the rmM-tree.

**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 dispositivos IoT, o crescimento dos datacenters e do serviço em nuvem vêm contribuindo fortemente para o aumento na produção de dados (?).

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

- 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 ?, 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 conectados. A International Data Corporation (IDC) prevê que em 2025, 75% (6 bilhões) da população mundial interaja com aplicações diversas todos os dias, fazendo com que a quantidade de dados produzidos cresça de 33 Zetabytes (ZBs) em 2018, para 175 ZBs em 2025 (?).

Esse crescimento na 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 (?). Essa disparidade é um grande desafio para projetistas e desenvolvedores, pois como afirmam ?, 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 (??).

Como mostra ?, estruturas de dados sucintas são uma forma de compressão de dados. Elas 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 algoritmos de compactação 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 em que a quantidade de memória é limitada, como dispositivos IoTs e embarcados (?).

Com os dados residindo em memória principal, temos um novo gargalo, dessa vez entre memória cache e memória principal, nesse sentido e tendo em vista a lacuna de desempenho crescente entre processador e memória RAM, faz-se necessário buscarmos formas de melhor utilizarmos a cache. ? implementaram em um de seus trabalhos, diferentes versões de uma estrutura denominada *Cache-Sensitive Search Trees (CSS-Tree)*, a *Cache-Sensitive  $B^+$ -Tree (CSB<sup>+</sup>-Tree)*, a proposta do trabalho deles era melhorar o desempenho de operações de atualização existentes na CSS-tree, maximizando a quantidade de dados em cada nó. Durante os experimentos foi constatado que, a estrutura original possuía melhor desempenho nas operações de pesquisa devido ao seu alto fator de ramificação, se comparado ao da CSB<sup>+</sup>-tree. Para as operações de atualização, as diferentes versões da CSB<sup>+</sup>-tree tiveram melhor desempenho em relação à CSS-tree. Entre as diferentes versões da CSB<sup>+</sup>-tree implementadas, os autores obtiveram melhor desempenho, tanto em operações de atualização, como de pesquisa, nas versões que tinham fator de ramificação mais alto.

Nesse trabalho, nos concentraremos no estudo de árvores sucintas, uma das estruturas mais populares no campo da Ciência da Computação, tanto no caso das representações clássicas, quanto para o caso das representações sucintas. De acordo com ?, 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 ancestral comum mais baixo de dois nós, até aquelas que suportam um conjunto completo de operações, podendo variar em relação ao espaço ocupado, indo de  $O(n/(\log \log n)^2)$  à  $O(n/\text{polylog}(n))$  bits.

A nossa contribuição consiste na proposta de uma versão, de uma estrutura de dados sucinta já existente: a range min-Max tree (rmM-tree), de [1]. Essa estrutura fornece suporte à diversas operações sobre árvores gerais 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  $O(\log n)$  [1]. Essa estrutura é construída no formato de uma árvore binária completa, e portanto possui um baixo fator de ramificação. Assim, com base no exposto por [1], propomos neste trabalho uma versão da rmM-tree binária, no formato de uma árvore k-ária, afim de maximizar o fator de ramificação da range min-Max tree. Propomos também não só um fator de ramificação maior para essa estrutura, como também um número maior de entradas por nó, aumentando assim o volume de dados enviados a cache, em cada transferência.

## 1.1 Objetivos

O objetivo central deste trabalho é a construção da proposta de uma versão alternativa da range min-Max tree, visando diminuir a quantidade de transferências de dados necessárias entre memória principal e cache, melhorando assim o desempenho geral da estrutura. Como objetivos secundários, temos: estudo e compreensão das representações sucintas de árvores, seus fundamentos e benefícios; compreensão dos impactos da hierarquia de memória e os benefícios da utilização adequada dos recursos fornecidos por esta arquitetura.

## 1.2 Metodologia

Esse estudo será feito com base na revisão bibliográfica da estrutura proposta por [1], visando melhor entendimento das operações de navegação e consulta em árvores gerais fornecidas pelo nosso objeto de estudo. Implementaremos a versão original da rmM-tree, buscando abranger todas as operações suportadas na literatura, além da versão original será implementada a versão alternativa da rmM-tree, no formato de árvore k-ária, com até k entradas por nó. Para realizar a validação das nossas implementações, usaremos a biblioteca *Succinct Data Structure Library (SDSL)*, uma biblioteca construída em C++, com código aberto e de alta qualidade, que reúne os destaques de 40 publicações de pesquisa [2]. Auxiliará na validação das respostas das estruturas implementadas o framework Google Tests. Já o desempenho das estruturas será analisado mediante o uso do framework Google Benchmark.

## 1.3 Estrutura do documento

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, ao final deste capítulo trazemos uma discussão acerca do aproveitamento efetivo da cache. O capítulo ?? apresenta a nossa versão da rmM-tree k-ária, expondo suas principais características e diferenças em relação à estrutura clássica. O capítulo ??, discorre a respeito



dos testes experimentais, metodologia usada para os mesmos e resultados alcançados. Por fim o capítulo ?? expõe as nossas conclusões em relação aos objetivos e resultados alcançados, bem como as nossas perspectivas em relação aos trabalhos futuros.

## 2

### Referencial teórico

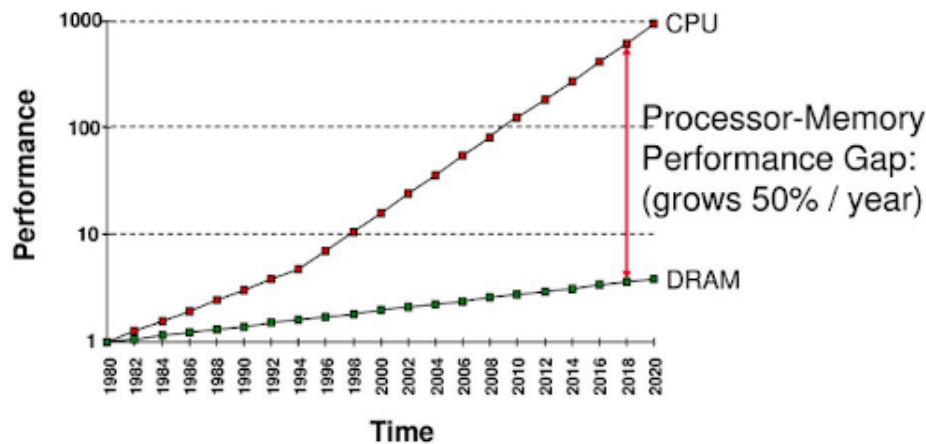
Quando trabalhamos com um conjunto de dados relativamente grande, parte deste tende a ser distribuído entre os níveis da hierarquia de memória, podendo acarretar em um alto número de transferência de dados entre os níveis da memória e consequentemente numa piora do desempenho geral do sistema.

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. É apresentando também os conceitos básicos das representações que são amplamente usados na construção dessas estruturas. Tratamos também, acerca da range min-Max tree (rmM-tree) que serve como objeto desse estudo, por fim apresentamos a literatura que serviu como motivação para a construção da nossa proposta.

#### 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 –. Isto 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, esses diferentes objetivos fazem com que o desempenho desses dois componentes cresça com uma diferença de 50% ao ano (??), como ilustrado na Figura ???. 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’ (? , 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 ?? demonstra uma hierarquia de memória, seus respectivos valores de latência<sup>1</sup> e espaço para cada nível de um servidor e de um dispositivo móvel hipotético. Através dessa figura é possível observar que, à medida que nos distanciamos do processador, as unidades de tempo variam de centenas de picossegundos até milissegundos, ao passo que a capacidade de armazenamento varia de bytes para terabytes.

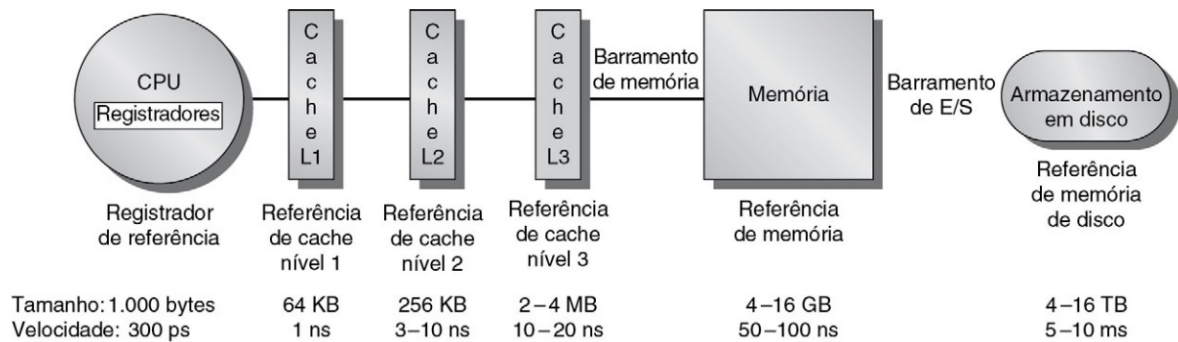
Devido a alta latência que as memórias mais distantes do processador possuem, torna-se ideal trabalhar nos níveis superiores da hierarquia de memória para obter um desempenho efetivo das operações sobre um conjunto de dados (?). A grande problemática é que, conforme disposto por ?, as memórias com latência menor possuem capacidade de armazenamento também menor, tornando praticamente inviável manipular grandes conjuntos de dados nestes níveis, o que é cada vez mais necessário nos dias atuais, devido ao aumento crescente na produção e consumo de dados. Uma possível solução para tanto é operar sobre os dados em sua forma compactada, e conforme cita também ?, a melhor maneira de fazer isso é através do uso das estruturas de dados sucintas. Possuindo relação direta com a Teoria da Informação<sup>2</sup>, uma estrutura de dados é dita sucinta se ela pode representar informações usando um espaço próximo à entropia<sup>3</sup> determinada por essa teoria, que como mostra ?, 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 manipulação de grandes conjuntos de dados em memórias como a cache e a RAM, ao passo que

<sup>1</sup>Tempo entre o início e conclusão de uma solicitação do processador a memória.

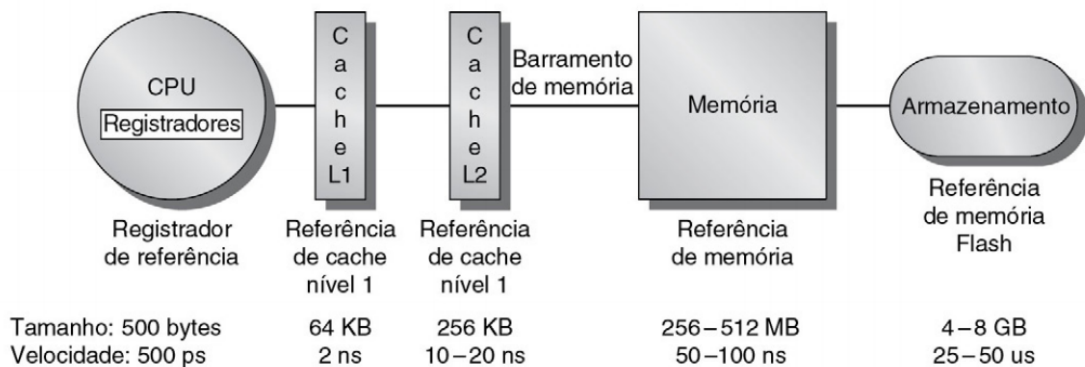
<sup>2</sup>Teoria matemática proposta inicialmente por Claude Shannon que busca quantificar a informação.

<sup>3</sup>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: ?

nas estruturas de dados sucintas, não é necessário descompatar a informação para operar sobre ela.

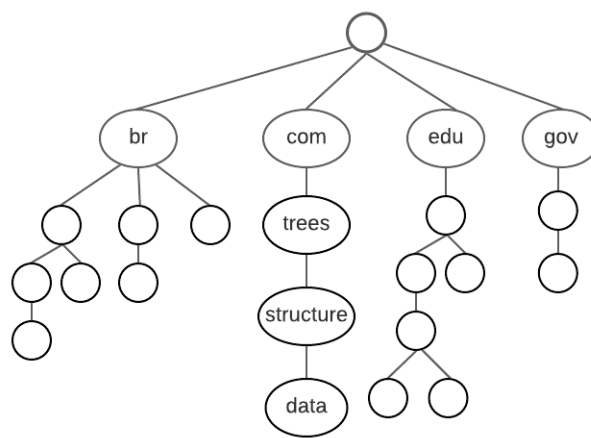
## 2.2 Árvores Sucintas

‘Sendo uma das estruturas de dados mais difundidas na computação, as árvores também são uma das histórias de sucesso mais marcantes nas estruturas de dados sucintas’ (? , tradução nossa). 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 (?).

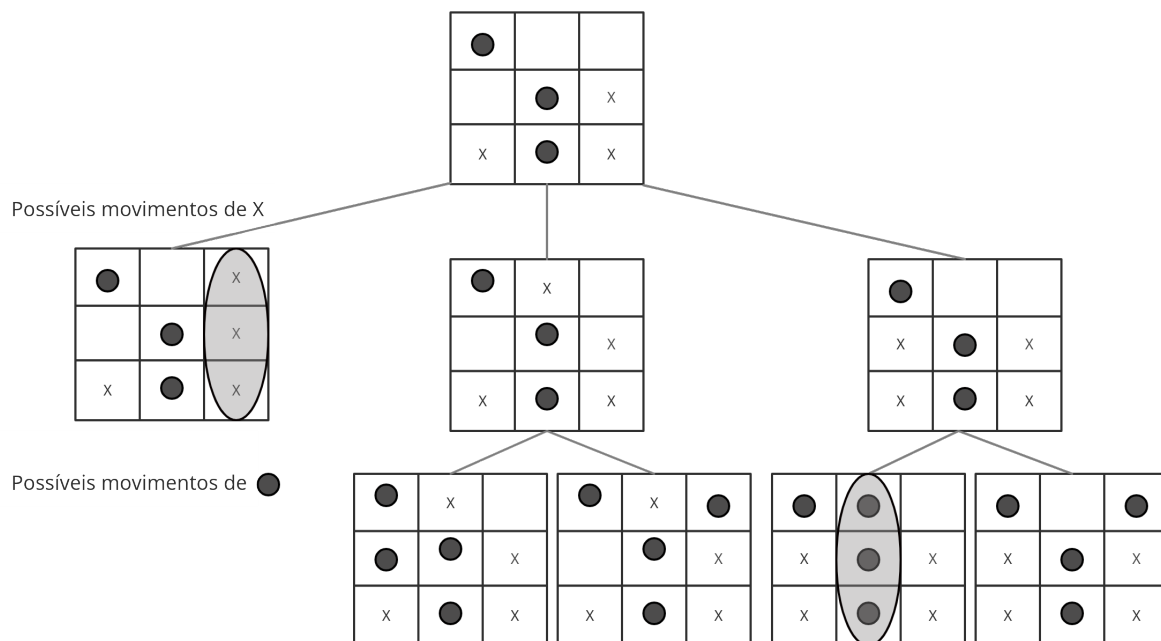
Encontramos diversas aplicações construídas a partir destas estruturas, entre elas aplicações de banco de dados e tradutores de idiomas (?). Árvores também são amplamente usadas no campo de aprendizagem de máquina: são as chamadas árvores de decisão, que auxiliam desde a escolha do melhor movimento em um jogo de xadrez à operações em sites de comércio eletrônico (?). Outra aplicação das árvores no mundo real, se dá através do mapeamento de *Domain Name System (DNS)* em endereços *IP* (e virse-versa). Como mostra (?) a árvore de

*DNS* representa de modo hierárquico os nomes de um domínio, facilitando a sua administração e escalonamento. O *DNS* é armazenado de forma invertida na árvore, e cada nó possui um rótulo (de até 63 caracteres), representando um *host* ou *subdomínio*, a raiz, por padrão, recebe um rótulo nulo, e o nome de qualquer domínio pode ser obtido a partir do percurso de um nó folha até chegar ao nó raiz. É possível ver um exemplo de uma árvore de *DNS*, que armazena o domínio *data.structure.trees.com* na Figura ??, essa figura também exemplifica parte uma árvore de decisão usada em um jogo da velha.

**Figura 2.3:** Exemplos de representação gráfica de árvores



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



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

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, 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 que viabiliza operações de montagem de genomas. Com 3,3 bilhões de nucleotídeos, se usarmos 2 bits para armazenar cada base nitrogenada necessitaremos 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 em 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<sup>4</sup>, enquanto existem representações sucintas, abordadas nas Seções ?? e ??, que ocupam cerca de  $2n + o(n)$  bits.

### 2.3 Vetores de Bits

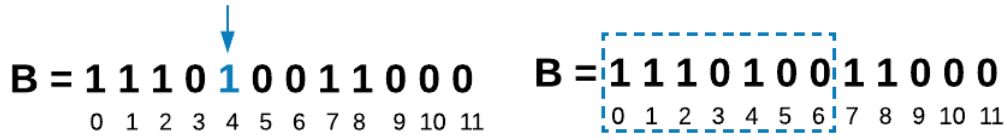
Um vetor de bits  $BV[0, n-1]$  é uma sequência sobre o alfabeto  $\Sigma = \{0, 1\}$ . É interessante que as seguintes operações sejam executadas sobre os vetores de bits (?):

- $access(BV, i)$ : retorna o  $i$ -ésimo bit do vetor  $BV$ , com  $0 \leq i < n$ ;
- $rank_v(BV, i)$ : seja  $v \in \{0, 1\}$ , e  $0 \leq i < n$ , esta operação retorna o número de ocorrências de  $v$  no intervalo  $BV[0, i]$ .  
Sendo que a seguinte relação de equivalência é válida:  $rank_0(BV, i) = i + 1 - rank_1(BV, i)$ ;
- $select_v(BV, i)$ : dado  $v \in \{0, 1\}$ , com  $0 \leq i < n_v$ , e sendo  $n_v$  o número máximo de ocorrências de  $v$  em  $BV$ ,  $select$  retorna a posição do  $i$ -ésimo bit  $v$  em  $BV[0, n-1]$ .

A Figura ?? traz exemplos das operações listadas.

### 2.4 Representação de árvores sucintas

Existem diversas formas de representar uma árvore de maneira sucinta, abaixo listamos algumas destas.

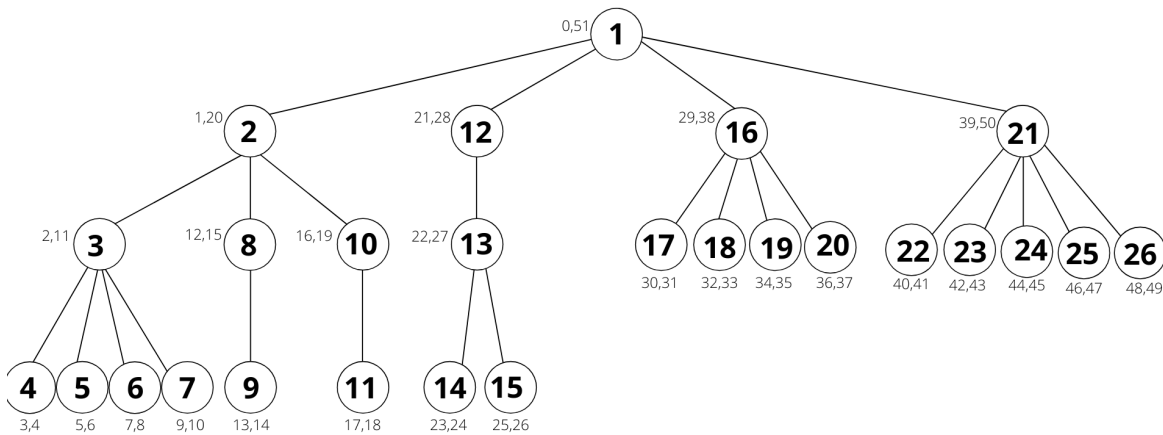
**Figura 2.4:** Operações de *rank*, *select* e *access* sobre  $B = 111010011000$ 

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

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



(c)  $select_0(BV, 5) = 8$

**Figura 2.5:** Representação de uma árvore  $T$  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 (?)

(((((())())())())())((())())((())())((())())())((())())((())())((())())())  
 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

### 2.4.1 Parênteses Balanceado (BP)

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 uma árvore ordinal<sup>5</sup>  $T$  usando essa estrutura, tomamos um vetor de bits

<sup>4</sup>Neste trabalho, quando não indicado, estaremos trabalhando com o logaritmo na base 2.

<sup>5</sup>Em uma árvore ordinal, cada nó pode ter um número arbitrário de filhos, e as subárvores de cada nó formam um conjunto ordenado de nós (?)

de tamanho igual à  $2n$  ( $BP[0, 2n - 1]$ ), em que  $n$  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 (podendo ser representado pelo bit 1) é inserido em  $BP$ , ao término da exploração das subárvores deste nó, um parênteses de fechamento (bit 0) é adicionado em  $BP$  (?). Na figura Figura ?? pode-se observar uma árvore com 26 nós e sua representação equivalente em parênteses balanceados.

Usando a representação de parênteses balanceados, podemos fornecer suporte às operações descritas 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  $i$ ’. (? , tradução nossa)

? proporam uma solução em árvores binárias usando a representação de parênteses balanceados, nesta proposta, além das operações já citadas, os autores viabilizaram suporte a operação de  $enclose(i)$ , que retorna o pai de um nó codificado em  $i$ , e a operação  $double\_enclose(i, j)$  (que equivale à operação  $lca$  em árvores) que por sua vez retorna o ancestral comum mais baixo dos nós  $i$  e  $j$ .

Com base na operação  $rank$  podemos viabilizar a operação de  $excess$  facilmente, como mostrado abaixo.

$$\begin{aligned} excess(BP, i) &= rank_1(BP, i) - rank_0(BP, i) \\ &= rank_1(BP, i) - (i - rank_1(BP, i)) \\ &= 2 \cdot rank_1(BP, i) - i - 1 \end{aligned}$$

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  $i$ .

Levando em consideração que em  $BP$  um nó é codificado através de um parênteses de abertura, podemos responder facilmente se um nó codificado  $j$  é ou não, um nó folha, para tanto, basta averiguar se o elemento que segue o índice  $j$ , codifica um novo nó, em caso afirmativo, o nó codificado por  $j$  possui filhos, e portanto não pode ser um nó folha, caso o elemento armazenado em  $j + 1$  corresponda à um parênteses de fechamento significa que o nó  $j$  não possui filhos, e portanto é um nó folha. Esta verificação pode ser feita através da operação de  $access(BP, j + 1)$ .





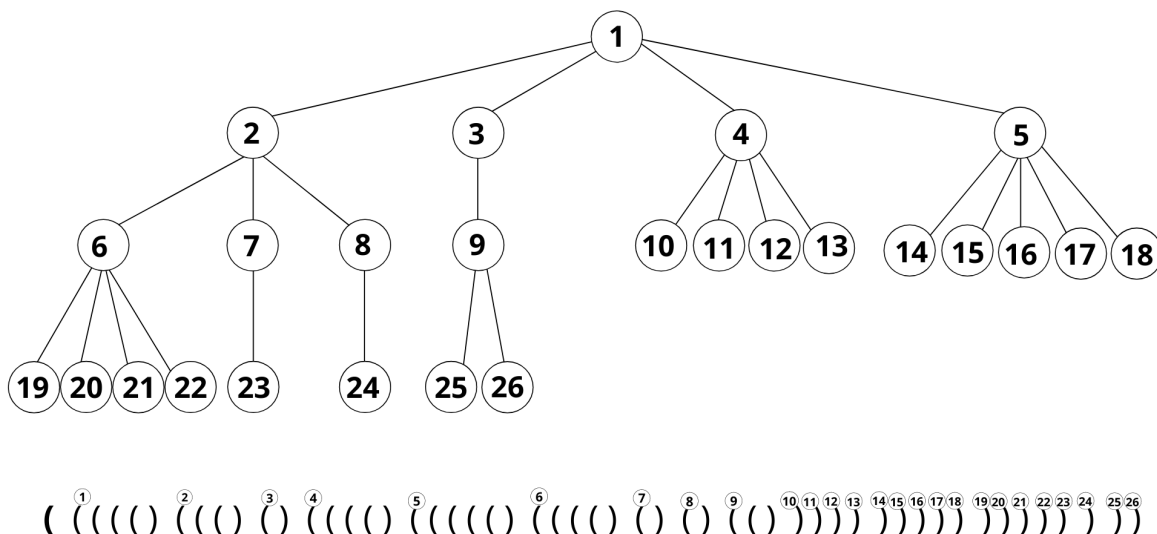
ser delimitada por um parênteses de fechamento, como acontece para *parênteses balanceados* (?).

### 2.4.3 Level-order Unary Degree Sequence (LOUDS)

Nesta representação, percorremos os nós da árvore em largura, adicionando assim como em *DFUDS*,  $i$  (novamente com  $i$  representando o grau do nó) parênteses de abertura ao visitarmos um nó pela primeira vez, seguido de 1 parênteses de fechamento. Como os nós são codificados através de um percurso em largura na árvore de entrada, essa estrutura não permite que as subárvores de um nó sejam mapeadas de forma contígua no vetor que a representa, inviabilizando operações básicas como o cálculo do tamanho da subárvore de um nó (??).

A figura abaixo mostra uma representação em *LOUDS* para a árvore  $T$ , usada como exemplo na representação de *parênteses balanceados* e em *DFUDS*.

**Figura 2.7:** Representação de uma árvore  $T$  usando LOUDS. Assim como em DFUDS foi adicionado um parênteses de abertura no início da sequência para que a mesma se tornasse balanceada.



## 2.5 Range min-Max Tree (rmM-tree)

Proposta por [?], estudada por diversos outros autores, a range min-Max tree, é uma estrutura baseada em valores de excessos máximos e mínimos dentro de um intervalo. Esta é construída com base em uma sequência de parênteses balanceados, e permite a redução de um grande número de operações relevantes sobre árvores consideradas na literatura, a poucas operações primitivas, podendo estas serem realizadas em tempo constante para árvores suficientemente pequenas [?].

Neste trabalho usaremos a abordagem mostrada por ? em seu livro, para a construção da rmM-tree binária. Como explica o autor, a range min-Max tree é construída na forma de uma

árvore binária completa –o que nos permite a utilização de vetores, omitindo o uso de ponteiros explícitos – sendo que cada um de seus nós armazena os valores de excesso calculados dentro de um intervalo do vetor que representa a árvore de entrada.

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

Como cada nó folha de uma rmM-tree está associado a um único intervalo, o tamanho de uma rmM-tree está também relacionado 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 via parênteses balanceados de uma árvore  $T$  com tamanho igual à  $n/2$ , é definido um tamanho de intervalo  $b$  (chamado também de tamanho de bloco), divide-se então  $n$  por  $b$ , obtendo assim 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  $BP$  ocupa  $n$  bits, temos assim que a rmM-tree utiliza um espaço próximo à  $n + O(\frac{n}{b} \log n)$  bits. Tomando  $b = \log^2 n$ , temos uma complexidade de espaço igual à  $n + O(n/\log n)$ . Em relação ao tempo gasto por cada operação sobre a rmM-tree, ? mostra que na prática todas essas operações podem ser realizadas em tempo  $O(\log n)$  - ou ainda em tempo  $O(\log \log n)$  em uma versão mais refinada da estrutura - a lista completa dessas operações é apresentada na Tabela ?? deste capítulo.

### 2.5.1 Registros

Os valores de excesso definidos 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, ?, 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  $BP$  definido anteriormente, então:

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

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

- $R[v].m$ : corresponde ao excesso mínimo local.

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

- $R[v].M$ : corresponde ao 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 valor do excesso mínimo no intervalo  $[s, e]$ , então:

$$R[v].n = |\{s \leq i \leq e \mid excess(i) = R[v].m\}|$$

Agora que temos a definição de cada registro da range min-Max tree, podemos construir os nós folhas da nossa estrutura, e os demais nós nível a nível, partindo das folhas. Os nós internos e raiz da nossa árvore, são calculados a partir dos valores de seus nós filhos, como a nossa estrutura é construída em um vetor, podemos navegar até os filhos de um nó  $v$  usando aritmética básica nos índices da nossa árvore, desse modo, o filho esquerdo de um nó  $v$  é dado por  $(2 \cdot v) + 1$ , ao passo que o filho direito desse nó pode ser obtido a partir de  $(2 \cdot v) + 2$ . Essa relação entre os valores dos registros de um nó pai e um nó filho, é descrita pelas equações abaixo:

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

**Exemplo 1:** Para melhor elucidar a construção dessa estrutura usaremos o exemplo da Seção ?? e demonstraremos o cálculo dos registros e dos intervalo de um nó folha, mostraremos também como é construído um dos nós internos da rmM-tree. No exemplo em questão, temos um vetor de entrada com 52 parênteses balanceados, e 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  $2 \cdot r - 2^h - 1 = 9$  estão agrupadas no último nível da árvore, e as outras  $2^h - r =$  folhas estão agrupadas à direita no nível anterior.

Assim temos os seguintes valores de excesso para a folha 0 (nó 15) da rmM-tree:

- Área de cobertura:  $BP[0, 3]$
- Excesso local:  $R[15].e = excess(3) = 2 \cdot rank_1(BP, 3) - 3 - 1 = 4$

- Excesso mínimo:  $R[15].m = \min(1, 2, 3, 4) = 1$
- Excesso máximo:  $R[15].M = \max(1, 2, 3, 4) = 4$
- Número de vezes que o excesso mínimo ocorre no intervalo:  
 $R[15].n = \text{count}(1, 2, 3, 4) = 1$

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:  $BP[0, 3] \cup BP[4, 7]$
- 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 raiz, dando origem a árvore mostrada na Figura ??.

O processo de construção da rmM-tree é mostrado com mais detalhes no Algoritmo ??. Neste algoritmo, fazemos o uso de uma tabela  $C$ , esta tabela pré-computa todas as possibilidades de valores de registro para um intervalo tamanho  $w$ , sendo  $b \bmod w = 0$ . ‘Essa técnica de pré-computar todos os valores possíveis de determinados padrões de bits é conhecida como *truque dos quatro russos*’ (? , tradução nossa). O uso dessa tabela  $C$  não é obrigatório para a construção da rmM-tree, entretanto, a mesma possibilita a otimização do tempo de construção e operação sobre a rmM-tree, como disposto em ?.

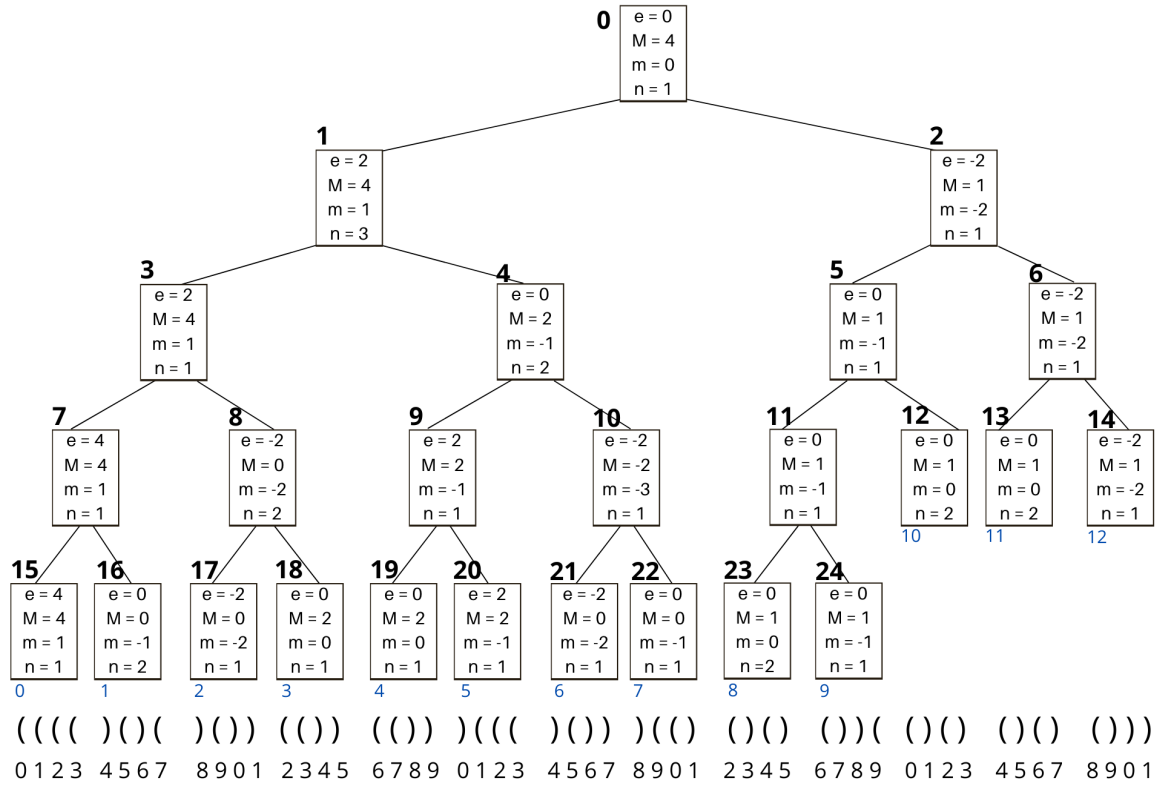
Conforme expõe ?, a tabela  $C$  é construída em tempo  $O(\sqrt{n} \log n)$  e ocupa  $O(\sqrt{nw})$  bits de espaço. Esse processo de construção é feito do seguinte modo: após definir uma constante  $w$ , criamos uma tabela  $C[0, 2^w - 1]$ , onde cada entrada da tabela armazena os registros de excesso definidos anteriormente, cada elemento  $C[x]$  é calculado com base nos  $w$  bits que formam  $x$ . Abaixo segue um exemplo explicando a computação de uma entrada específica da tabela  $C$ .

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

$$3_2 = 0011,$$

logo iterando sobre os bits que formam o número temos os seguintes valores de registro:

**Figura 2.8:** rmM-tree clássica, com tamanho de bloco igual à 4. A estrutura foi construída a partir dos 52 parênteses balanceados mostrados na parte inferior



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

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 da rmM-tree, ou durante a inspeção dos mesmos, basta ler os  $w$  bits que queremos inspecionar, e então realizar uma consulta na tabela  $C$  pelo elemento  $x$  que mapeia os bits lidos. Conforme mostra ? uma consulta na tabela  $C$  leva tempo constante, já a construção dos nós folhas usando essa técnica 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 então lê a palavra formada a partir de  $i$  seguido pelos  $w - 1$  bits após  $i$ , usando como critério de leitura o bit mais significativo (*Most Significant Bit (MSB)*). O valor obtido por *bitsread* é usado para consultar a tabela  $C$ , em busca dos valores de registro 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 (veja Algoritmo ??).

**Exemplo 3:** Tome como base agora, a nossa rmM-tree de exemplo, assuma  $w = 4$ . Para melhor exemplificação, vamos alterar um pouco a estrutura da nossa árvore, assuma um tamanho de bloco  $b = 8$  para este exemplo. Suponha que queremos construir a folha de número 0 (como aumentamos a cobertura do nosso bloco, a folha 0 neste exemplo corresponde à união dos nós 15 e 16 da árvore mostrada na Figura ??), que neste exemplo corresponde ao nó 7. O intervalo coberto por essa folha vai de 0 à 7, assim precisaríamos ler os 8 bits que compõem 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 tamanho de bloco também maior, o processo se tornaria mais oneroso. Observe então como este processo é feito usando a tabela de excessos  $C$ .

- Como  $w = 4$ , convertamos primeiro os bits codificados em  $BP[0, 3]$   
 $BP[0, 1, 2, 3] = 1111_2$   
 Fazemos então uma consulta na tabela  $C$ , pelo elemento  $x = 1111_2$ , que com base na explicação anterior, traz os seguintes valores:

$$C[1111_2].e = 4; C[1111_2].M = 4; C[1111_2].m = 1; C[1111_2].n = 1.$$

Nesse momento, os valores do registro da nossa folha são:

$$R[7].e = 4; R[7].M = 4; R[7].m = 1; R[7].n = 1.$$

- Agora precisamos computar os valores correspondentes aos  $w$  bits restantes que compõem a nossa folha, temos que:  $BP[4, 5, 6, 7] = 0101_2$ . Os registros armazenados em  $C$ , para este elemento são:

$$C[0101_2].e = 0; C[0101_2].M = 0; C[0101_2].m = -1; C[0101_2].n = 2.$$

Agora que temos os valores de excesso para cada subintervalo da nossa folha, podemos obter os valores de registro para o intervalo completo, esse processo é similar ao definido para a obtenção dos registros de um nó pai. Temos assim que:

- $R[7].e = R[7].e + C[0101_2].e = 4 + 0 = 4$
- $R[7].M = \max(R[7].M, R[7].e + C[0101_2].M) = \max(4, 4 + 0) = 4$
- $R[7].m = \min(R[7].m, R[7].e + C[0101_2].m) = \min(1, 4 - 1) = 1$
- $R[7].n = 1$ , pois o excesso mínimo ocorre no primeiro subintervalo da folha.

Ao observar o nó 7 do exemplo ??, é possível notar, que de fato, os valores obtidos nestes exemplo correspondem ao nó citado.

**Algoritmo 1:** Conversão entre número de folha e índice de folha na rmM-tree.

---

```

1 Proc numLeaf( $v$ )
    Input: Índice ( $v$ ) da rmM-tree.
    Output: Número ( $l$ ) da folha codificada em  $R[v]$ .
2   if  $v \geq 2^h - 1$  then
3     return  $v - 2^h + 1$ 
4   end if
5   else
6     return  $v - 2^h + r + 1$ 
7   end if
1 Proc leafInTree( $l$ )
    Input: Número ( $l$ ) de uma folha na rmM-tree.
    Output: Índice ( $v$ ) onde a folha é codificada na rmM-tree.
2   if  $l < (2r - 2^h)$  then
3     return  $2^h - 1 + l$ 
4   end if
5   else
6     return  $2^h - 1 - r + l$ 
7   end if

```

---

**2.5.2 Operações**

As operações sobre a range min-Max tree são realizadas através de cálculos usando os valores de excesso definidos na seção anterior. Parte dessas já foram descritas nas Seções ?? e ??, como as operações *enclose*, *findclose* e *findopen*, nesta seção veremos como computá-las usando os nós da rmM-tree. Além das operações citadas, existem outras importantes operações suportadas pela range min-Max tree, detalhamos algumas logo abaixo.

A Tabela ?? mostra a lista completa das operações suportadas pela 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 em um índice  $i$  no vetor que representa uma árvore geral. Desse modo, *fwdSearch* retorna um índice  $j > i$ , mais à esquerda possível, 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:

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

Como veremos mais tarde, dessa operação deriva-se diversas outras, tais como *findclose*, *lca* (*ancestral comum mais baixo*) e outras operações de percurso em árvore.

? descrevem o funcionamento dessa operação da seguinte maneira: dado um excesso



---

**Algoritmo 2:** Construção da range min-Max tree binária

---

```

1 Algoritmo buildingTree(BP, C, b, w, r, nNodes)
   Input: Vetor de bits, tabela de excessos C, tamanho de bloco e de subbloco,
           número de folhas e quantidade de nós.
   // Construção dos nós folhas
2 for  $l \leftarrow 0$  to  $r - 1$  do
3    $v \leftarrow \text{leafInTree}(l)$  // Algoritmo ??
4    $(R[v].e, R[v].M, R[v].m, R[v].n) \leftarrow (0, -w, w, 0)$ 
5   for  $p \leftarrow (l \cdot (b/w)) + 1$  to  $((l + 1) \cdot b)/w$  do
6      $x \leftarrow \text{bistread}((p - 1) \cdot w)$ 
7     if  $R[v].e + C[x].M > R[v].M$  then
8        $R[v].M \leftarrow R[v].e + C[x].M$ 
9     if  $R[v].e + C[x].m < R[v].m$  then
10       $R[v].m \leftarrow R[v].e + C[x].m$ 
11       $R[v].n \leftarrow 1$ 
12     else if  $R[v].e + C[x].m = R[v].m$  then
13        $R[v].n \leftarrow R[v].n + C[x].n$ 
14      $R[v].e \leftarrow R[v].e + C[x].e$ 
   // Construção dos nós internos e raiz
15 for  $v \leftarrow nNodes - r - 1$  to  $0$  do
16    $v_l \leftarrow (2 \cdot v) + 1$ 
17    $v_r \leftarrow v_l + 1$ 
18    $R[v].e \leftarrow R[v_l].e + R[v_r].e$ 
19    $R[v].M \leftarrow \max(R[v_l].M, R[v_l].e + R[v_r].M)$ 
20    $R[v].m \leftarrow \min(R[v_l].m, R[v_l].e + R[v_r].m)$ 
21   if  $R[v_l].m > R[v_l].e + R[v_r].m$  then
22      $R[v].n \leftarrow R[v_r].n$ 
23   else
24      $R[v].n \leftarrow R[v_l].n + R[v_r].n$ 

```

---

desejado  $d$ , e um índice  $i$ , a partir do qual a busca dever ser feita, examinamos a folha  $k$  (com  $k = \lfloor (i+1)/b \rfloor$ ), cujo o intervalo de cobertura engloba o índice  $i+1$ . Caso não encontremos  $d$  dentro desse intervalo, usamos os nós da rmM-tree para encontrar o bloco a qual  $d$  pertence. Avançamos na rmM-tree pelo pai do nó folha analisado, verificando a cada iteração, se o valor do excesso buscado está compreendido no intervalo dos valores de excesso do nó à direita do atual. Ao encontramos o nó que contém o valor de excesso buscado, encerramos o processo de subida na árvore, e iniciamos o processo de descida na rmM-tree até que cheguemos à um nó folha. Durante o processo de descida, avançamos pelo filho à esquerda do nó corrente sempre que  $d$  estiver incluso nos intervalos de contenção do mesmo, e pelo nó à direita caso contrário. Ao chegarmos em um nó folha da árvore, interrompemos a inspeção dos nós da rmM-tree e iniciamos um escaneamento dos bits que compõem o intervalo dessa folha.

O exemplo ?? demonstra o processo acima, o Algoritmo ?? fornece mais detalhes de como *fwdSearch* é computada.

**Exemplo 4:** Dado um nó em *BP*, codificado pelo parênteses de abertura localizado em  $i = 21$ , encontrar o limite desse nó, representado pelo seu parênteses de fechamento em  $BP[i]$ .

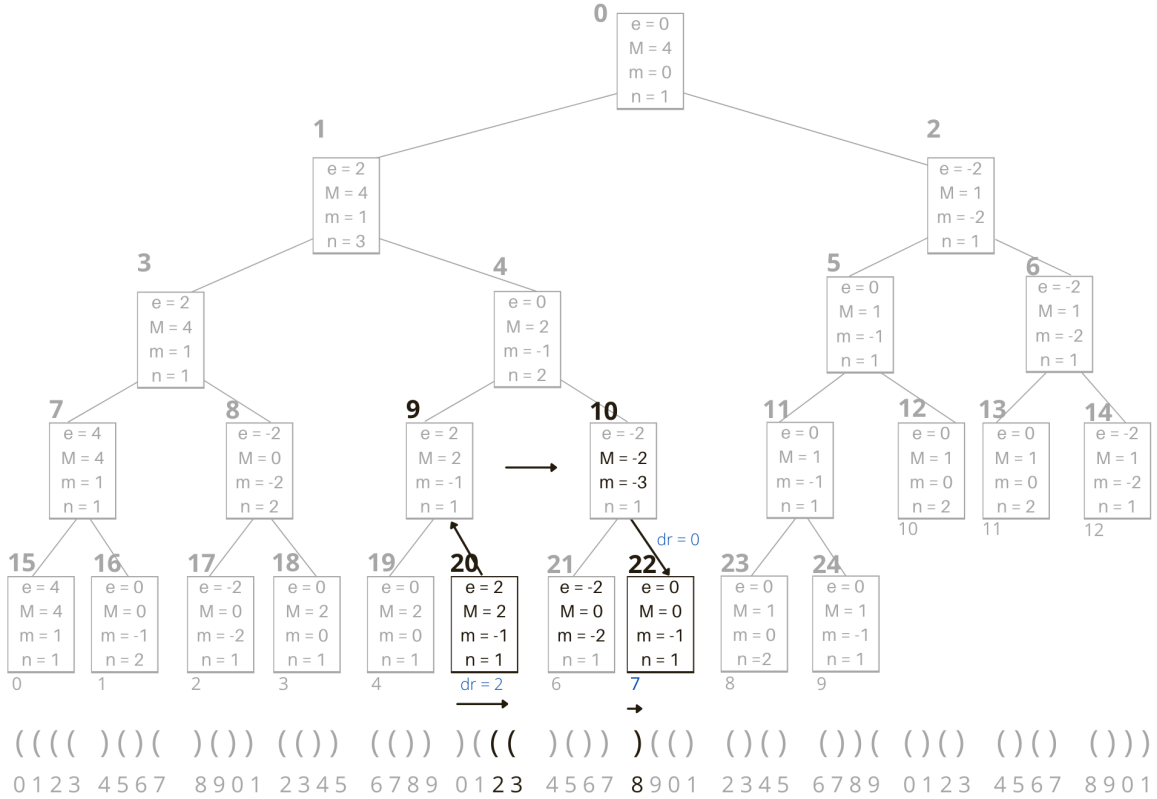
Para encontrarmos o parênteses de fechamento de um nó, basta realizarmos uma busca a partir de  $i$  pelo excesso  $d = -1$ . Ou seja, queremos encontrar  $j > i$ , mais à esquerda possível, tal que  $excess(j) - excess(i) = -1$ . Essa operação é facilmente respondida por *fwdSearch*( $i, -1$ ).

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

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 chegar ao fim do intervalo deste nó que é  $j = 23$ , durante esse processo de inspeção simulamos a operação de  $rank_1$ , guardando em  $dr$  os valores computados. Ao fim do intervalo, temos que  $dr = 2$  (como apontado na figura), como o valor buscado não foi encontrado nessa varredura inicial, precisamos aumentar o nosso espaço de busca. Fazemos isso subindo na rmM-tree através dos seus nós.

? traz um passo interessante que otimiza a busca na árvore, essa técnica consiste em verificar se a resposta buscada está no nó à direita do atual, evitando subidas/descidas desnecessárias na rmM-tree. Com base nessa técnica, analisamos o nó à direita de 20, sem obter sucesso. Atualizamos  $v$  para o índice do seu nó pai (9), e verificamos se os campos do nó à direita (nó 10) adicionados de  $dr$ , compreendem o excesso relativo buscado  $d$ , ou seja  $2 - 3 \leq 1 \leq -2 + 2$ . Essa asserção é válida, portanto interrompemos a subida na árvore, e atualizamos o índice guardado em  $v$  para 10.

Iniciamos o percurso de descida na árvore. Como buscamos a posição  $j$  mais à esquerda possível de  $i$ , verificamos primeiro se a resposta está contida no intervalo

**Figura 2.9:** Simulação da operação  $fwdSearch(21, -1)$  em uma rmM-tree binária.

do filho esquerdo do nó 10, o que não é verdade, em decorrência disso adicionamos à  $dr$  o excesso local do nó 21 ( $dr + (-2) = 0$ ), e descemos então pelo filho direito de  $v$ , e nesse momento chegamos à um nó folha.

Através desse procedimento de descida na árvore, reduzimos ao máximo o intervalo a ser inspecionado. Iniciamos então a inspeção do intervalo coberto pelo nó 22, temos que,  $dr = -1$  em  $BP[28]$ , que é o excesso buscado, desse modo temos que 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. 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, tal que:

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

---

**Algoritmo 3:** Busca pelo excesso relativo  $d$  em um nó folha.

---

```

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

2    $fb \leftarrow \lceil (i+1)/w \rceil$ 
3    $lb \leftarrow \lceil (i+2)/b \rceil \cdot (b/w)$ 
4   for  $j \leftarrow i+1$  to  $(lb \cdot w) - 1$  do
5       if  $BP[j] = 1$  then
6            $dr \leftarrow dr + 1$ 
7       else
8            $dr \leftarrow dr - 1$ 
9       if  $dr = d$  then
10          return  $j$ 
11  for  $p \leftarrow fb + 1$  to  $lb$  do
12       $x \leftarrow \text{bitsRead}((p-1) \cdot w)$ 
13      if  $dr + C[x].m \leq d \leq dr + C[x].M$  then
14          break
15       $dr \leftarrow dr + C[x].e$ 
16  if  $p > lb$  then
17      return  $lb \cdot b //$   $dr$  não está no bloco subsequente
18  for  $j \leftarrow (p-1) \cdot w$  to  $(p \cdot w) - 1$  do
19      if  $BP[j] = 1$  then
20           $dr \leftarrow dr + 1$ 
21      else
22           $dr \leftarrow dr - 1$ 
23      if  $dr = d$  then
24          return  $j$ 
25  return BP.size()

```

---

---

**Algoritmo 4:** Busca por um excesso relativo  $d$ , para um  $j > i$ 


---

```

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

2    $dr \leftarrow 0$ 
3    $j \leftarrow fwdBlock(i, d, \&dr)$  // Algoritmo ??
4   if  $dr = d$  then
5     return  $j$ 
6    $l \leftarrow \lfloor (i+1)/b \rfloor$ 
7    $v \leftarrow leafInTree(l)$  // Algoritmo ??
8   while  $(v+1) \& (v+2)$  and  $dr + R[v+1].m \leq d \leq dr + R[v+1].M$  do
9     if  $v \bmod 2$  then
10       $dr \leftarrow dr + R[v+1].e$ 
11       $v \leftarrow \lfloor (v-1)/2 \rfloor$ 
12   if  $(v+1) \& (v+2) = 0$  then
13     return  $BP.size()$ 
14    $v \leftarrow v+1$ 
15   while  $v < numberLeaves - 1$  do
16      $v_l \leftarrow (2 \cdot v) + 1; v_r \leftarrow v_l + 1$ 
17     if  $dr + R[v_l].m \leq d \leq dr + R[v_l].M$  then
18        $v \leftarrow v_l$ 
19     else
20        $dr \leftarrow dr + R[v_l].e$ 
21        $v \leftarrow v_r$ 
22    $l \leftarrow numLeaf(v)$  // Algoritmo ??
23    $j \leftarrow fwdBlock((l \cdot b) - 1, d, \&dr)$ 
24   if  $dr = d$  then return  $j$ ;
25   else return  $BP.size()$ ;

```

---

Como explica ??, 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 ( $dr$ ), no intervalo  $[i, j]$ , seja igual ao excesso buscado,  $d$ . Mas como  $j < i$  e portanto  $dr = d = -excess(j+1, i)$ , devemos adicionar 1 unidade à  $dr$  ao inspecionarmos um bit codificado como 0, e diminuir  $dr$  em 1 unidade caso contrário;
- Os valores de excesso são computados a partir de um processo de inspeção de bits da esquerda para a direita. A operação *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.

Ademais, como estamos fazendo uma busca por uma posição  $j < i$ , mais à direita possível, durante o percurso na rmM-tree inspecionamos primeiramente o filho à direita de um nó em busca da resposta, caso não encontremos esta, avançamos no percurso através do filho à esquerda. O Algoritmo ?? detalha mais sobre esse processo, este algoritmo invoca um método denominado *bwdBlock*, este por sua vez é completamente análogo ao Algoritmo ?? (a única ressalva é a questão de simetria discutida acima), portanto não fornecemos aqui o pseudo-código de *bwdBlock*.

O exemplo a seguir, ilustra o processo de busca pelo índice que codifica um nó, cujo parênteses de fechamento é armazenado em  $BV[i]$ .

**Exemplo 5:** Dado um índice  $i$  em  $BP$ , igual à 50, encontrar o índice  $j < i$ , mais à direita possível, tal que  $excess(j) - excess(i) = 0$

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

A nossa busca inicia a partir da análise dos bits compreendidos no intervalo da folha 12 (nó 14) que compreende o índice  $i = 50$  (reafirmamos que *bwdSearch* inclui o índice  $i$ ), ao realizarmos a inspeção desta folha, indo de  $i = 50$  até  $j = 49$ , temos um excesso relativo computado até o momento ( $dr$ ) igual à 1.

Como o excesso é diferente do valor buscado precisamos usar os nós da rmM-tree para encontrar a resposta. Usamos a mesma técnica de otimização de *fwdSearch* e



---

**Algoritmo 5:** Busca por um excesso relativo  $d$ , para um  $j < i$ .

---

```

1 Algoritmo 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

2    $dr \leftarrow 0$ 
3    $j \leftarrow \text{bwdBlock}(i, d, \&dr)$  // Análogo ao algoritmo ??
4   if  $dr = d$  then
5     return  $j$ 
6    $l \leftarrow \lfloor i/b \rfloor$ 
7    $v \leftarrow \text{leafInTree}(l)$ 
8   while  $(v+1) \& v$  and
      $dr - R[v-1].e + R[v-1].m \leq d \leq dr - R[v-1].e + R[v-1].M$  do
9     if  $v \bmod 2 = 0$  then
10       $dr \leftarrow dr - R[v-1].e$ 
11       $v \leftarrow \lfloor (v-1)/2 \rfloor$ 
12   if  $(v+1) \& v = 0$  then
13     return  $-1$ 
14    $v \leftarrow v-1$ 
15   while  $v < \text{numberLeaves} - 1$  do
16      $v_l \leftarrow (2 \cdot v) + 1; v_r \leftarrow v_l + 1$ 
17     if  $dr - R[v_r].e + R[v_r].m \leq d \leq dr - R[v_r].e + R[v_r].M$  then
18        $v \leftarrow v_r$ 
19     else
20        $dr \leftarrow dr - R[v_r].e$ 
21        $v \leftarrow v_l$ 
22    $l \leftarrow \text{numLeaf}(v)$ 
23    $j \leftarrow \text{bwdBlock}(((l+1) \cdot b) - 1, d, \&dr)$ 
24   if  $dr = d$  then
25     return  $j$ 
26   else
27     return  $-1$ 

```

---

### 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 da inspeção do nó folha que contém o índice  $i$ . Para *minExcess*, a folha é analisada a partir do índice  $i$  até chegarmos ao seu limite, ou até visitarmos o índice  $j$ , caso este esteja englobado no intervalo da folha de  $i$ .



Durante a análise da folha, a cada bit lido, guardamos em uma variável  $d$  o valor do excesso relativo computado até o momento, verificamos também se este excesso é o menor até o momento.

Terminando a inspeção da folha que contém  $i$ , caso  $j$  não esteja contido nela, iniciamos o processo de subida na árvore (caso  $j$  esteja contido, a busca é encerrada, e é retornado o menor excesso computado no intervalo analisado). Durante o percurso pela rmM-tree verificamos se os registros de excesso mínimo salvo em cada nó é menor que o excesso mínimo computado até o momento, em caso afirmativo, atualizamos o excesso mínimo computado até o momento. E independente da resposta, atualizamos o valor do excesso relativo  $d$ .

Ressalta-se que o percurso em árvore para esta operação é idêntico ao percurso em árvore em *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 informações, trazidas por  $\text{?}$ , que são:

- A profundidade de qualquer nó em  $R[v]$  é dada por  $\lfloor \log(v+1) \rfloor$ ;
- O pai de um nó, é dado por  $\lfloor (v-1)/2 \rfloor$ ;
- À medida que avançamos da esquerda para a direita na rmM-tree, seguindo a abordagem bottom-up, um nó  $u$  é dito ancestral de um nó  $v$ , se a asserção abaixo for válida,

$$\lfloor (v-1)/2^{\lfloor \log(v+1) \rfloor - \lfloor \log(u+1) \rfloor} \rfloor = u$$

- Conforme explica o autor, durante o percurso na árvore, a asserção acima pode ter um comportamento inesperado quando  $u$  for um nó folha, e tiver uma profundidade maior que a do nó alvo (a folha que contém o índice  $j$ ).  $\text{?}$  explica que 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. Desse modo, caso o índice do nó  $u$  não for maior do que o índice do nó alvo, e a asserção do tópico anterior for verdadeira, a busca é interrompida.

Ao encontrarmos o ancestral do nó alvo, interrompemos a subida na rmM-tree, e começamos o processo de descida a partir deste ancestral. O percurso de descida ocorre de modo similar ao da operação *fwdSearch*, verificamos primeiro se o filho esquerdo do nó visitado é ancestral do nó alvo, em caso positivo descemos na rmM-tree pelo filho esquerdo de  $v$ . Em caso negativo, usamos os registros de excesso deste nó para verificar se existe um excesso mínimo menor que o computado até o momento, atualizando esse valor conforme a resposta que obtivermos; e então atualizamos o valor de  $v$  para o índice do filho à direita deste nó.

Repetimos todo esse processo até que cheguemos a um nó folha. Tendo atingido um nó folha, realizamos a inspeção do mesmo, do seu início, até a posição  $j$ , verificando se os



nó alvo, e obtivemos que o nó 10 não é ancestral de  $x$ , com base nos registros do nó 10 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  $x = 24$ , e nesse momento temos uma afirmação verdadeira.

Tendo achado o nó ancestral de  $x$ , interrompemos o processo de subida na árvore, e iniciamos o processo de descida a partir do nó  $v = 5$ . Ao analisarmos os filhos de  $v$ , temos que o seu filho esquerdo (nó 11) é ancestral do nó 24, então atualizamos o valor de  $v$  para 11, e verificamos os filhos do mesmo. Temos nesse momento que o filho esquerdo de  $v$  (nó 23) não é ancestral do nó alvo, desse modo atualizamos o valor de  $v$ , para o índice do seu filho à direita, que é o próprio nó alvo.

Como chegamos à um nó folha, interrompemos a verificação dos nós da rmM-tree, e iniciamos a verificação do excesso mínimo na folha 9 (nó 24), ao percorrer está folha o valor do excesso mínimo computado até o momento não é alterado, e portanto  $BP[18, 39].m = -3$ .

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 excesso mínimo, verificamos o excesso máximo).

### 2.5.6 MinCount e MinSelectExcess

Essas operações possuem bastante similaridade entre si, e também são análogas à *minExcess*, portanto não entraremos em detalhes sobre as duas.

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 percorrer a rmM-tree em busca da resposta esperada para cada operação.

Para realizar a operação *minCount*, cria-se um acumulador que é incrementado através do registro  $n$ , da rmM-tree, sempre que passarmos por um nó cujo o valor de excesso mínimo for igual ao excesso computado por *minExcess*.

A operação *minSelectExcess* é análoga, para ela, pode-se criar um acumulador que recebe o valor  $t$ , passado para a função, e assim ao visitar um nó cujo o valor de excesso mínimo é igual ao valor computado por *minExcess*, subtraímos do acumulador o valor armazenado em  $n$ , a resposta nesse caso é encontrada quando o acumulador atinge a marca menor ou igual à 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 registro  $n$  da rmM-tree, portanto não há necessidade de fazer a computação deste, 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 primeiro parâmetro em  $fwdSearch$  é subtraído em 1, isso acontece porque o intervalo da busca de  $fwdSearch$  é aberto, ou seja, não inclui o índice  $i$  passado nas buscas, ao passo que o intervalo de  $rmq$  é fechado, incluindo  $i$  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 nó, isto é, a um parênteses de abertura, e a partir da operação  $fwdSearch$ , busca o parênteses de fechamento correspondente. A operação  $fwdSearch$  percorrerá o vetor de bits e rmM-tree a partir de  $i + 1$  até que encontre 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* retornará o pai do nó  $x$ , usando a operação *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 operação buscará pelo nó  $y$  que seja ancestral e esteja  $d$  níveis acima de um nó  $x$ . Para tanto podemos usar a operação *bwdSearch*. Por questões de simetria, já discutidas anteriormente, devemos passar o valor de  $d$  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 usar *findClose*, como mostrado abaixo:

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

- **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 descendente 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  $i$ . Como a sequência que representa a nossa árvore de entrada é balanceada, sabemos que dentro do intervalo (aberto) de hierarquia de um nó, atingimos o excesso mínimo sempre que finalizamos a codificação de um de seus nós filhos. Levando isso em consideração, podemos calcular  $degree$  do seguinte modo:

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

- **childRank:** dado um índice  $i$ ,  $childRank$ , contabiliza o número de irmãos que o nó codificado em  $i$  possui à sua esquerda. Tendo as operações definidas anteriormente, podemos responder à esta operação conforme mostrado abaixo:

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

- **nextSibling:** dado um nó codificado em  $i$ ,  $nextSibling$  busca pelo irmão localizado no mesmo nível do nó  $i$ , codificado em  $j$ , tal que  $j > i$ .

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

- **prevSibling:** análoga a  $nextSibling$ , esta operação busca pelo irmão imediatamente à esquerda do nó codificado por  $i$ , em seu mesmo nível. 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 essa operação busca pelo índice  $j$  onde acontece a  $t$ -ésima ocorrência do 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 enraizada 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 a direita do nó codificado por  $i$ , em seu mesmo nível:

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

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

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

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

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

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

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

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 estas 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 codificado em  $i$ , verifica se este é um nó folha ou não, essa é uma das operações mais simples da nossa estrutura e necessita apenas de 2 comparações,

veja abaixo:

$$isLeaf(i) = (BP[i] = 1 \wedge BP[i+1] = 0)$$

- **depth**: computa a profundidade de um nó codificado por  $i$ , para tanto basta verificar o excesso 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 ocorrências 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 a folha mais à esquerda da subárvore com raiz em  $i$ .

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

- **rightMostLeaf**: retorna a folha mais à direita da subárvore com raiz em  $i$

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

- **preRank**: retorna o índice do nó na árvore de entrada correspondente à  $i$ , considerando um percurso em pré-ordem.

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

- **postRank**: retorna o índice do nó na árvore de entrada correspondente à  $i$ , considerando um percurso em pós-ordem.

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

- **preSelect**: retorna a posição em  $BP$ , do  $i$ -ésimo nó na árvore de entrada, considerando



um percurso em pré-ordem:

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

- **postSelect**: retorna a posição em *BP*, do *i*-ésimo nó na árvore de entrada, considerando um percurso em pós-ordem:

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

**Tabela 2.1:** Operações suportadas pela rmM-tree binária

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

## 2.6 rmM-tree k-ária

Como vimos, o uso das estruturas de dados sucintas permitem que grandes conjuntos de dados residam e sejam operados na memória principal. ? nos mostram que, com os dados residindo na memória RAM, um novo gargalo é criado, que surge em decorrência do atraso ao buscarmos na memória principal um dado que não se encontra em cache.

Quando uma referência de memória é satisfeita pela cache, estas prosseguem na velocidade do processador, as referências insatisfeitas incorrem em uma penalidade no tempo de

execução da busca, devido a necessidade de buscar o bloco correspondente na memória principal. Conforme citam ? o tempo gasto para buscar um dado em cache é de uma à duas ordens de magnitude menor do que a latência para a recuperação de dados na memória principal. Desse modo, a utilização inadequada da cache pode fazer com que a latência da memória se torne um gargalo crescente de desempenho, impedindo que as aplicações se beneficiem das crescentes melhorias de hardware (?).

? realizaram um estudo sobre o efeito da maximização da quantidade de informações relevantes armazenada em um nó de uma árvore denominada *Cache-Sensitive Search Trees (CSS-tree)*, uma estrutura de índice consciente de cache, que possui um desempenho em realização de pesquisas superior ao da pesquisa binária citeppaper-making-btree-cache. De acordo com os autores o custo de execução de uma pesquisa é baseado em quatro fatores: número de instruções ( $I$ ), quantidade de *cache misses* ( $CM$ ), número de previsão erradas de ramificação ( $R$ ) e número de erros de TLB <sup>7</sup> ( $T$ ) (?). Esse tempo é modelado pela equação abaixo,

$$t = (I \cdot cpi) + (CM \cdot miss\_latency) + (R \cdot pred\_penalty) + (T \cdot tlb\_penalty)$$

sendo:

- $t$  o tempo total de execução em ciclos de clock;
- $cpi$ : custo de execução por instrução;
- $miss\_latency$ : tempo acrescido em decorrência de uma falta de cache;
- $pred\_penalty$ : custo de percorrer um ramo na árvore sem encontrar resposta, e
- $tlb\_penalty$ : custo para recuperar uma entrada na tabela de páginas.

Além disso ? nos mostram que o número de perdas de cache é limitado pela altura da árvore, essa por sua vez está diretamente relacionada ao número de nós folhas, e ao fator de ramificação da árvore. Com base nisso, parte da estratégia dos autores consistiu em aumentar a quantidade de dados coberto por nó folha, visando minimizar a altura da árvore, e fazendo melhor aproveitamento de uma linha de cache<sup>8</sup>.

Os resultados do trabalho citado nos mostra que, ao escolhermos um número de entradas adequado para um nó (não ultrapassando o tamanho de uma linha de cache), temos um número de *cache misses* mínimo. Além disso, os resultados experimentais de ? revelam que, mesmo ao maximizar o tamanho de um nó, ao ponto de ultrapassar o tamanho de uma linha de cache, o número de instruções e de previsões erradas, assim como a quantidade de páginas *TLB*, reduzem de modo tão drástico que compensa a quantidade de falhas de cache, superando o desempenho

<sup>7</sup>Translation Lookaside Buffer - armazena as entradas das tabelas de páginas da memória usadas recentemente

<sup>8</sup>Unidade básica de transferência entre cache e memória RAM

da versão implementada pelos autores da *CSS-tree* com tamanho de nó menor e número de *cache misses* também menor.

? também fizeram um estudo sobre o impacto de diferentes versões da *CSS-tree* em cache, o objetivo deles era melhorar o desempenho das operações de atualização suportadas pela *CSS-tree*. Com base nisto, os autores propuseram uma estrutura baseada em árvores  $B^+$  – tendo em vista que estas possuem excelente desempenho para conjuntos de dados que requerem atualizações incrementais– denominada *Cache Sensitive  $B^+$ -tree* ( $CSB^+$ -tree).

No total, ? criaram 3 versões da *CSS-tree*, são elas:  $CSB^+$ -tree,  $CSB^+$ -tree segmentada e *Full  $CSB^+$ -tree*, estas se diferenciam em quantidade de ramificações, capacidade de armazenamento de nó, e contiguidade de nós. Os resultados experimentais do estudo revelaram que: todas as versões da  $CSB^+$ -tree possuem desempenho superior ao da  $B^+$ -tree, tanto para pesquisa quanto para atualização, devido ao maior fator de ramificação das  $CSB^+$ -tree. Como esperado pelos autores, as *CSS-tree*, apresentam desempenho superior ao das  $CSB^+$ -tree para operações de pesquisa, isso se deve também ao elevado fator de ramificação da estrutura clássica se comparado aos das diferentes versões da  $CSB^+$ -tree. Em relação as 3 versões da  $CSB^+$ -tree, obtiveram melhor desempenho nas operações de pesquisa, àquelas que possuíam maior fator de ramificação. Por fim, as  $CSB^+$ -tree apresentou, conforme o objetivo do trabalho, melhor desempenho nas operações de atualização frente à *CSS-tree* (?).

Com base no exposto acima, e levando em consideração que a range min-Max tree possui baixo fator de ramificação e baixo aproveitamento de linha de cache. Propomos uma versão alternativa da rmM-tree, a rmM-tree k-ária. Nosso objetivo com essa estrutura é realizar um melhor aproveitamento da cache, maximizando a quantidade de dados transferidos entre níveis, e aumentando o fator de ramificação dessa estrutura, essa proposta é melhor detalhada no próximo capítulo.

### 3

#### Range min-Max tree k-ária

Este capítulo refere-se ao objetivo central deste trabalho, apresentando a proposta de modificação da estrutura criada por ?. Mostraremos primeiro no que consiste essa adaptação, após isso apresentaremos a forma como a rmM-tree k-ária deve ser construída, apresentando também as operações suportadas por esta estrutura. Destaca-se também que os exemplos usados como base neste capítulo, são 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  $k$  filhos, as árvores B possuem altura igual à  $\log_k n$ , 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, como discutido no capítulo ??.

Dado uma representação compacta de uma árvore geral  $T$ , na forma de parênteses balanceados, é escolhido um tamanho de bloco  $b$ . Esse tamanho de bloco, assim como na estrutura clássica define o tamanho de cobertura de cada intervalo compreendido por um registro. Na proposta original ?, cada nó da range min-Max tree é responsável pela cobertura de um desses intervalos, nesta adaptação propõe-se o agrupamento múltiplo de intervalos em nós.

Com base nisso, após definir uma aridade  $k$ , para a rmM-tree, seguindo a mesma abordagem *bottom-up* da rmM-tree binária, construímos os nós folhas da nossa estrutura, armazenando até  $k$  registros <sup>1</sup> (que cobrem intervalos de tamanho  $b$ ) de valores de excesso por nó folha da nossa estrutura. Terminando de construir os nós folhas, iniciamos a construções dos internos da range min-Max tree k-ária, nível a nível, partindo do penúltimo nível da árvore. Cada nó  $v$  da árvore será construído a partir da união de seus  $k$  filhos, sendo que cada registro de um nó  $v$  cobrirá a união dos registros do nó  $k \cdot v + 1 + i_{registro}$ . Isso implica que teremos até  $k^2$  áreas cobertas por nó (pois  $k$  filhos  $\cdot (k)$  registros), dessa maneira temos o exposto: se na versão clássica da rmM-tree

---

<sup>1</sup> As árvores B, armazenam no máximo  $k - 1$  chaves por nó, em nossa estrutura limitamos o número de registros de um nó à  $k$

cada nó folha armazena no máximo um intervalo, e portanto possui  $r = \lceil n/b \rceil$  folhas, na versão k-ária teremos  $r = \lceil n/(b \cdot k) \rceil$  folhas.

Como a altura de uma árvore pode ser calculada a partir de  $h = \lceil \log_k r \rceil$  essa simples mudança reduzirá drasticamente a altura da nossa estrutura, quando comparada a rmM-tree clássica, potencialmente reduzindo o número de eventuais transferências de dados entre cache e RAM.

Tomemos como exemplo um tamanho de bloco  $b = 32$ , suponha um vetor de bits que codifica um árvore geral  $T$ , de tamanho igual à 4.675.776.358, e que a nossa rmM-tree k-ária tem ordem igual à 16. A tabela abaixo ajuda a esclarecer o exposto acima, mostrando a altura e a quantidade de folhas para cada versão da rmM-tree.

**Tabela 3.1:** Altura e número de nós folhas para uma rmM-tree binária e k-ária

Tipo	Tamanho da árvore de entrada	Quantidade de folhas da rmM-tree	Altura da rmM-tree
binária	4.675.776.358	$\lceil 4.675.776.358/32 \rceil = 146.118.012$	$\lceil \log_2(146.118.012) \rceil = 28$
16-ária	4.675.776.358	$\lceil (4.675.776.358/(32 \cdot 16)) \rceil = 9.132.376$	$\lceil \log_{16}(4.711.244) \rceil = 6$

Ao aumentarmos o fator de ramificação de uma árvore, tiramos proveito do princípio de localidade espacial, que afirma que itens cujos endereços estão próximos um do outro tendem a ser referenciados em um curto espaço de tempo (?), diminuindo assim o número de eventuais transferência de dados entre cache e memória RAM.

### 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 em um 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 agrupados em até  $k$  blocos de tamanho  $b$  por folha, isso nos dá às seguintes relações para os nós internos e raiz:

- $R[v][rg].e = \sum_{i=0}^{l-1} R[j][i].e$
- $R[v][rg].m = \min(R[j][0].m, \dots, R[j][0].e + \dots + R[j][l-1].e + R[j][l].m)$
- $R[v][rg].M = \max(R[j][0].M, \dots, R[j][0].e + \dots + R[j][l-1].e + R[j][l].M)$
- $R[v][rg].n = \text{número de vezes que o excesso mínimo computado aparece nos registros de } j.$

em que:

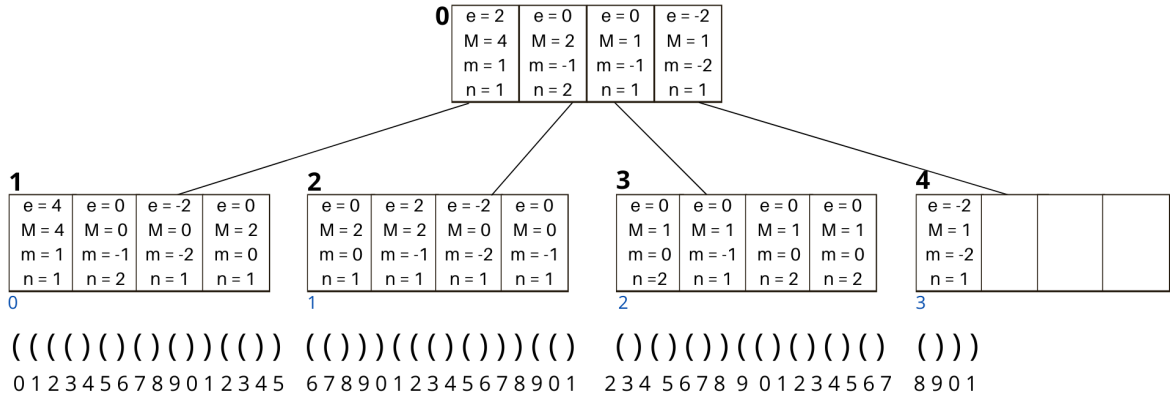
- $rg$ , é o  $rg$ -ésimo registro de  $v$  e aponta para o nó  $j = (v \cdot k) + 1 + rg$  (com  $0 \leq rg < k$ );
- $j$  é o  $j$ -ésimo filho de  $v$ ;
- $l$  é a quantidade de registros em  $j$ .

**Exemplo 7:** Usando o exemplo ?? de sequência de parênteses balanceados da seção ??, assuma uma cobertura de bloco igual à 4, e uma rmM-tree de ordem também igual à 4. Com base nas definições anteriores, temos que a range min-Max tree terá 4 nós folhas, e no máximo 4 registros por nó, pois:

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

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

**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.



Iniciamos a construção da nossa árvore partindo dos nós folhas, como estamos utilizando o mesmo exemplo da seção ??, cada uma de nossas folhas será equivalente ao agrupamento de 4 folhas consecutivas do exemplo ?? (a nossa folha 0 é equivalente ao agrupamento das folhas 0, 1, 2 e 3 da estrutura binária, por exemplo). A altura da range min-Max tree  $k$ -ária, neste exemplo é 1, e portanto temos apenas o nó raiz acima das nossas folhas. Cada registro do nosso nó raiz, cobrirá a união dos registros dos seguintes nós:

- $R[0][0]$  cobre a união dos registros do nó 1, pois:  $(0 \cdot 4) + 1 + 0 = 1$ .  
 $R[0][0].e = R[1][0].e + R[1][1].e + R[1][2].e + R[1][3].e = 2$ ;

$R[0][0].M = 4$ , pois:

$$\begin{aligned}
 R[0][0].M &= \max(R[1][0].M, \\
 &\quad R[1][0].e + R[1][1].M, \\
 &\quad R[1][0].e + R[1][1].e + R[1][2].M, \\
 &\quad R[1][0].e + R[1][1].e + R[1][2].e + R[1][3].M) \\
 &= \max(4, 4, 4, 4) = 4;
 \end{aligned}$$

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

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

$R[0][0].n = 1$ .

- $R[0][1]$  cobre os registros do nó 2, pois:  $(0 \cdot 4) + 1 + 1 = 2$ , e

$$R[0][1].e = 0; R[0][1].M = 2; R[0][1].m = -1 \text{ e } R[0][1].n = 2.$$

- $R[0][2]$  compreende a união dos registros do nó 3, pois:  $(0 \cdot 4) + 1 + 2 = 3$ , e

$$R[0][2].e = 0; R[0][2].M = 1; R[0][2].m = -1 \text{ e } R[0][2].n = 1.$$

- $R[0][3]$  cobre a união dos registros do nó 4, pois:  $(0 \cdot 4) + 1 + 2 = 4$ , e

$$R[0][3].e = -2; R[0][3].M = 1; R[0][3].m = -2 \text{ e } R[0][3].n = 1.$$

Observe que, se unirmos os intervalos que compõem os registros do nó 0, obteremos um registro equivalente ao do nó raiz mostrado no exemplo da construção da rmM-tree binária.

O processo de construção da rmM-tree k-ária é similar ao processo de construção da rmM-tree binária, descrito no capítulo ??, a diferença é que, para este caso precisaremos guardar em um nó até  $k$  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 os registros, de um nó  $v$  da rmM-tree, ao qual essa estrutura auxiliar está associada. O Algoritmo ?? ilustra o processo de construção da rmM-tree k-ária.

A tabela C, as funções *bitsread*, *leafInTree* e *numLeaf* – algumas citadas nesse e em outros algoritmos – seguem a mesma abordagem da rmM-tree clássica. O elemento *nReg* associado à um nó *v* corresponde, ao número de registros naquele nó.

### 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 forward search, realiza uma busca a partir de um índice *i*, afim de encontrar a posição  $j > i$ , 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, em busca do excesso desejado, a partir de  $i + 1$ . A diferença é que agora precisamos verificar até *k* blocos de tamanho *b* dentro de cada nó inspecionado na nossa estrutura. Para tanto, acrescentamos à *fwdSearch* em sua versão clássica duas outras funções, são elas *fwdRegistry* e *fwdVerifySibling*.

A primeira função é chamada sempre que estamos visitando um nó folha, ou seja no início da busca e ao final da mesma, quando já encontramos o nó alvo que contém a resposta. *FwdRegistry* tem por objetivo verificar se o excesso buscado está compreendido nos valores de excesso dos registro do nó visitado. O processo é muito similar à verificação de um nó na rmM-tree clássica: a cada registro percorrido verificamos a asserção  $dr + R[v][reg].m \leq d \leq dr + R[v][reg].M$ , caso seja verdadeira, significa que encontramos o intervalo que contém a resposta desejada. Se a asserção for falsa, adicionamos ao excesso relativo computado até o momento (*dr*), o excesso local do registro que acabamos de visitar, seguindo para o próximo registro.

O Algoritmo ?? fornece mais detalhes de como esse processo de busca por excesso nas folhas funciona.

Caso o excesso buscado não seja encontrado na folha inicial, acionamos o processo de subida na árvore, aliado à nova função que mencionamos, *fwdVerifySibling*. O objetivo dessa função é similar ao proposto por ?, quando o autor sugere que em cada nível verifiquemos se o excesso procurado está no irmão à direita do nó atual. Assim *fwdVerifySibling* calcula a quantidade de irmãos do nó *v* existentes à sua direita, e então analisa os registros de cada um deles em busca do intervalo que contém o excesso buscado, atualizando o valor de *dr* sempre que *d* não for encontrado em um registro.



**Algoritmo 6:** Construção da range min-Max tree k-ária

---

```

1 Algoritmo buildingTree(BP, C, k, b, w, r, nNodes)
   Input: Vetor de bits, tabela de excessos C, aridade da árvore, tamanho de bloco
           e de subbloco, número de folhas e quantidade de nós.
   // Construção dos nós folhas
2   numReg  $\leftarrow$  0
3   for l  $\leftarrow$  0 to r - 1 do
4       v  $\leftarrow$  leafInTree(l) // Análogo ao Algoritmo ??
5       reg  $\leftarrow$  0
6       while R[v].nReg < k and numReg <  $\lceil BP.size() / b \rceil$  do
7           (R[v][reg], R[v][reg].M, R[v][reg].m, R[v][reg].n)  $\leftarrow$  (0, -w, w, 0)
8           for p  $\leftarrow$  (numReg · (b/w)) + 1 to ((numReg + 1) · b) / w do
9               x  $\leftarrow$  bistread((p - 1) · w)
10              if R[v][reg].e + C[x].M > R[v][reg].M then
11                  R[v][reg].M  $\leftarrow$  R[v][reg].e + C[x].M;
12              if R[v][reg].e + C[x].m < R[v][reg].m then
13                  R[v][reg].m  $\leftarrow$  R[v][reg].e + C[x].m;
14              else if R[v][reg].e + C[x].m = R[v][reg].m then
15                  R[v][reg].n  $\leftarrow$  R[v][reg].n + C[x].n
16                  R[v][reg].e  $\leftarrow$  R[v][reg].e + C[x].e
17              R[v].nReg  $\leftarrow$  R[v].nReg + 1
18              numReg  $\leftarrow$  numReg + 1
19              reg  $\leftarrow$  reg + 1
20
21          // Construção dos nós internos e raiz
22          for v  $\leftarrow$  nNodes - r - 1 to 0 do
23              for reg  $\leftarrow$  0 to k - 1 and (v · k) + reg + 1 to nNodes - 1 do
24                  child  $\leftarrow$  (k · v) + 1 + reg
25                  R[v][reg].e  $\leftarrow$  R[child][0].e
26                  R[v][reg].m  $\leftarrow$  R[child][0].m
27                  R[v][reg].M  $\leftarrow$  R[child][0].M
28                  R[v][reg].n  $\leftarrow$  R[child][0].n
29                  for i  $\leftarrow$  1 to R[child].nReg - 1 do
30                      R[v][reg].M = max(R[child][i - 1].M, R[v][i - 1].e + R[child][i].M)
31                      R[v][reg].m = min(R[child][i - 1].m, R[v][i - 1].e + R[child][i].m)
32                      if R[v][reg].m < R[child][i - 1].e + R[child][i].m then
33                          R[v][reg].n  $\leftarrow$  R[child][i].n
34                      else if R[v][reg].m = R[child][i - 1].e + R[child][i].m then
35                          R[v][reg].n  $\leftarrow$  R[v][reg].n + R[child][i].n
36                          R[v][reg].e  $\leftarrow$  R[child][i].e

```

---

**Algoritmo 7:** Verificando os registros de um nó folha.

---

```

1 Algoritmo fwdRegistry(i, v, reg, l, d, &dr)
   Input: Índice em BP, nó v, folha e registro correspondente, a partir dos quais a
           busca deve ser feita, excesso relativo buscado, e excesso relativo
           computado em cada registro inspecionado (d e &dr).
   Output: Posição j ou BP.size() caso d não seja encontrado.

2   for reg to R[v].nReg - 1 do
3     if ( $dr + R[v][reg].m \leq d \leq dr + R[v][reg].M$ ) then
4       j  $\leftarrow$  fwdBlock(i, d, dr) // Algoritmo ??
5       if dr = d then
6         return j
7       dr  $\leftarrow$  dr + R[v][reg].e
8       i  $\leftarrow$  (k · l + reg + 1) · b - 1 // calcula o fim da registro
           atual
9       if dr = d then
10        return i
11  return BP.size()

```

---

Se ao final da inspeção dos irmãos de *v*, não encontrarmos o registro que contém a resposta, atualizamos o valor de *v* para o índice do seu nó pai, e verificamos os irmãos deste nó atualizado. Repetimos esse processo até que encontrarmos um registro que contenha o excesso relativo desejado.

Tendo encontrado o intervalo que contém a resposta desejada durante a subida na *rmM-tree*, iniciamos o processo de estreitamento desse intervalo, através do processo de descida na *rmM-tree*. Esta etapa é similar as demais, e consiste em percorrer os registros de um nó *v*, verificando se a asserção  $dr + R[v][reg].m \leq d \leq dr + R[v][reg].M$  é válida, atualizando o valor do excesso computado até o momento, sempre que necessário. Ao inspecionarmos um registro que contém a resposta buscada, atualizamos o valor de *v*, com base nesse registro, descendo mais um nível da árvore. Repetimos esse processo até que cheguemos à um nó folha, que é quando executamos novamente *fwdRegistry* para varrer um nó folha.

Os Algoritmos ?? e ?? mostram como o processo de descida na árvore funcionam, bem como o funcionamento completo de *fwdSearch*.

O exemplo a seguir é baseado no exemplo ??, e demonstra como podemos usar a operação *fwdSearch* para obtermos a demarcação de término de um nó.

**Exemplo 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 e o registro do nó folha que cobre *i* = 21 + 1 (lembrando que a busca é por um *j* > *i*), neste caso temos *v* = 2 (folha 1), e *reg* = 1,

---

**Algoritmo 8:** Busca o excesso relativo nos irmãos à direita de  $v$ .

---

```

1 Algoritmo fwdVerifySibling(& $v$ , & $dr$ ,  $d$ )
   Input: Nó  $v$ , excesso relativo buscado e excesso relativo computado até o
           momento e excesso buscado.
   Output: Registro ( $reg$ ) que contém a resposta, ou  $BP.size()$  caso o intervalo que
           contém  $d$  não seja encontrado.

2    $parent \leftarrow \lfloor (v - 1) / k \rfloor$ 
3    $n\_sibling \leftarrow v - (parent \cdot k)$ 
4    $v \leftarrow v + 1$ 
5   while  $n\_sibling$  to  $R[parent].nReg - 1$  and  $v$  to  $num\_nodes - 1$  do
6       for  $reg \leftarrow 0$  to  $R[v].nReg - 1$  do
7           if  $(dr + R[v][reg].m \leq d \leq dr + R[v][reg].M)$  then
8               if  $v \leq numberNodes - numberLeaves$  then
9                    $v \leftarrow (v \cdot k) + 1 + reg$ 
10                  return  $reg$ 
11               $dr \leftarrow dr + R[v][reg].e$ 
12              if  $dr = d$  then
13                  if  $v \leq numberNodes - numberLeaves$  then
14                       $v \leftarrow (v \cdot k) + reg + 2$ 
15                      return  $reg + 1$ 
16               $n\_sibling \leftarrow n\_sibling + 1$ 
17               $v \leftarrow v + 1$ 
18 return  $BP.size()$ 

```

---

**Algoritmo 9:** Busca por um excesso relativo  $d$ , para um  $j > i$ 


---

```

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

2    $dr \leftarrow 0$ 
3    $l \leftarrow \lfloor (i+1)/(b \cdot k) \rfloor$ 
4    $v \leftarrow leafInTree(l)$ 
5    $reg \leftarrow \lfloor ((i+1) - (l \cdot b \cdot k))/b \rfloor$ 
6    $j \leftarrow fwdBlock(i, d, dr)$  // Algoritmo ??
7   if  $dr = d$  then
8     return  $j$ 
9    $reg \leftarrow reg + 1$ 
10  if  $reg < R[v].nReg$  then
11     $j \leftarrow fwdRegistry((k \cdot l + reg) \cdot b - 1, v, reg, l, d, dr)$ 
12    if  $dr = d$  then return  $j$ ;

    // Inicia o processo de subida na rmM-tree
13  while  $v \neq 0$  and  $fwdVerifySibling(v, dr, d) = BP.size()$  do
14     $v \leftarrow \lfloor (v-1)/m \rfloor$ 
    // Chegamos ao nó raiz, e  $d$  não está em nenhum dos
    // seus filhos
15  if  $v = 0$  and  $reg = v.size()$  then
16    return  $v.size()$ 

    // Inicia descida em árvore
17  while  $v \leq numberNodes - numberLeaves$  do
18    for  $reg \leftarrow 0$  to  $R[v].nReg - 1$  do
19      if  $(dr + R[v][reg].m \leq d \leq dr + R[v][reg].M)$  then
20         $v \leftarrow (v \cdot k) + 1 + reg$ 
21         $reg \leftarrow 0$ 
22        break
23      else  $dr \leftarrow dr + R[v][reg].e$ ;

24   $l \leftarrow numLeaf(v)$ 
25   $i \leftarrow (k \cdot l \cdot chave) \cdot sizeBlock$  // Calcula o índice inicial da
    // folha que queremos inspecionar
26   $j \leftarrow fwdRegistry(i-1, v, 0, l, d, dr)$ 
27  if  $dr = d$  then return  $j$ ;
28  return  $BP.size()$ 

```

---



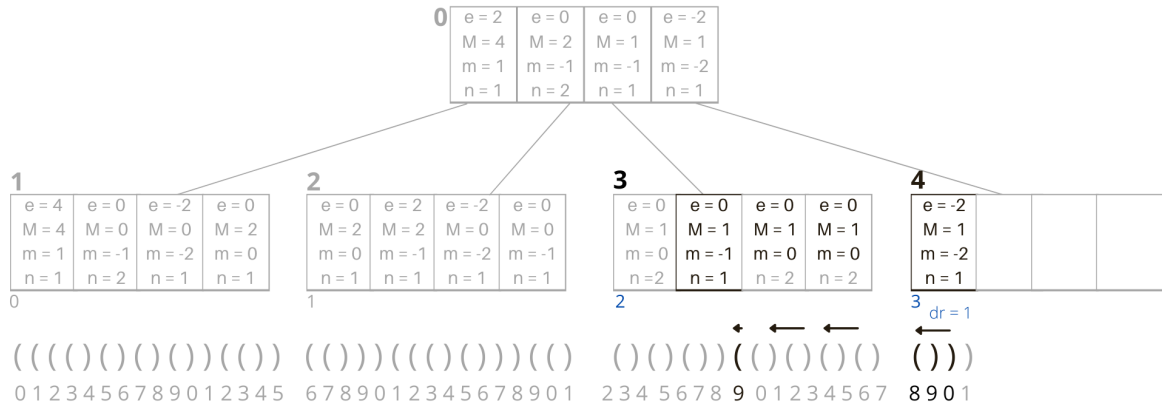
Apenas para recapitularmos, temos as seguintes observações em relação à assimetria dos dados em `bwdSearch`<sup>2</sup>:

- A posição  $i$  é incluída na contagem de excessos;
- Adicionamos 1 ao excesso relativo computado a cada inspeção, quando passamos por um bit 0, e subtraímos 1 desse excesso quando varremos um bit codificado como 1;
- $excess(i) + d$  é alcançado quando  $dr - R[v][reg].e + R[v][reg].m \leq d \leq dr - R[v][reg].e + R[v][reg].M$  for verdadeiro.

O exemplo ?? demonstra como podemos realizar a busca por um parênteses de abertura que codifica um nó, dado o seu respectivo parênteses de fechamento, através da operação `bwdSearch`. Este exemplo por sua vez, é equivalente ao exemplo ??.

**Exemplo 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)`.



Após identificarmos o nó e o registro a qual pertence o índice  $i = 50$ , temos que  $v = 4$  (folha 3) e  $reg = 0$  ( $\lfloor (50 - 48)/4 \rfloor$ ). Iniciamos a inspeção da folha 3, partindo de  $i = 50$ , até chegar ao início da mesma que é  $s = 48$ , ao término desta inspeção temos que  $dr = 1$ . Como o nó 4 não possui outros registros e a resposta não foi encontrada, realizamos a verificação dos registros localizados nos irmãos à esquerda do nó  $v$ .

Atualizamos  $v$  para o seu antecessor, que passa a valer 3, percorremos então cada um dos registros do nó  $v = 3$ , verificando se a asserção  $dr - R[3][reg].e + R[3][reg].m \leq d \leq dr - R[3][reg].e + R[3][reg].M$  é válida. Inspeccionamos os registros de número 3 e 2 do nó  $v$  sem encontrar a resposta, ao analisarmos o registro 1, temos que a

<sup>2</sup>Para mais detalhes, verificar seção ??

asserção definida é verdadeira, neste momento encerramos a verificação dos nós da rmM-tree e iniciamos uma varredura mais detalhada deste registro.

Ao realizarmos a inspeção dos bits que compõe o registro 1, do nó 3, encontramos  $dr = d$  em  $j = 49$ , finalizando desse modo a busca pelo índice em  $BV$  que codifica o parênteses de abertura correspondente à  $i = 50$ ;

Para este exemplo fizemos a inspeção de registros 3 (excluindo o registro inicial), já para a estrutura binária, como visto anteriormente, foi necessário, realizar a inspeção de 6 nós (excluindo o nó inicial), subir 1 nível da árvore, descendo depois 2 níveis.

---

**Algoritmo 10:** Verificando os registros de um nó folha.

---

```

1 Algoritmo bwdRegistry( $i, v, reg, l, d, \&dr$ )
   Input: Índice em  $BP$ , nó  $v$ , folha e registro correspondente, a partir dos quais a
           busca deve ser feita, excesso relativo buscado, e excesso relativo
           computado em cada registro inspecionado ( $d$  e  $\&dr$ ).
   Output: Posição  $j$  ou  $-1$  caso  $d$  não seja encontrado.

2   for  $reg$  downto 0 do
3     if  $(dr - R[v][reg].e + R[v][reg].m \leq d \leq dr - R[v][reg].e + R[v][reg].M)$  then
4        $j \leftarrow bwdBlock(i, d, \&dr)$  // Análogo ao algoritmo ??
5       if  $dr = d$  then
6         return  $j$ 
7      $dr \leftarrow dr - R[v][reg].e$ 
8      $i \leftarrow (k \cdot l + reg) \cdot b - 1$  // calcula o início da chave atual
9     if  $dr = d$  then
10      return  $i$ 
11  return  $-1$ 

```

---

### 3.3.3 Derivadas

Não entraremos nos detalhes das operações derivadas de *fwdSearch* e *bwdSearch*, tendo em vista que as mesmas já foram expandidas no capítulo ??.

A Tabela ?? traz um comparativo das operações suportadas pela rmM-tree k-ária e binária. Para a rmM-tree k-ária oferecemos suporte à um número menor de operações. Isso se deve ao fato de não implementamos as operações de *minExcess*, *maxExcess*, *minCount* e *minSelectExcess* que são base de muitas outras, destaca-se porém, que o processo de adaptação destas operações é similar ao que já foi exposto até aqui.

---

**Algoritmo 11:** Busca o excesso relativo nos irmãos à esquerda de  $v$ .

---

```

1  Algoritmo bwdVerifySibling(& $v$ , & $dr$ ,  $d$ )
   Input: Nó  $v$ , excesso relativo buscado e excesso relativo computado até o
           momento e excesso buscado.
   Output: Registro ( $reg$ ) que contém a resposta ou  $-1$  caso o intervalo que
           contém  $d$  não seja encontrado.

2   $parent \leftarrow \lfloor (v - 1) / k \rfloor$ 
3   $n\_sibling \leftarrow v - (parent \cdot k) - 1$ 
4   $v \leftarrow v - 1$ 
5  while  $n\_sibling$  downto 0 and  $v$  downto 0 do
6      for  $reg \leftarrow R[v].nReg - 1$  to 0 do
7          if  $(dr - R[v][reg].e + R[v][reg].m \leq d)$  and
             $(d \leq dr - R[v][reg].e + R[v][reg].M)$  then
8              if  $v \leq numberNodes - numberLeaves$  then  $v \leftarrow (v \cdot k) + 1 + reg$  return
                 $reg$ 
9               $dr \leftarrow dr - R[v][reg].e$ 
10             if  $dr = d$  then
11                 if  $v \leq numberNodes - numberLeaves$  then
12                      $v \leftarrow (v \cdot k) + reg$ 
13                 return  $reg$ 
14             if  $n\_sibling - 1 > 0$  then
15                  $v \leftarrow v - 1$ 
16              $n\_sibling \leftarrow n\_sibling - 1$ 
17 return  $-1$ 

```

---



---

**Algoritmo 12:** Busca por excesso no intervalo  $[i + 1, BP.size() - 1]$ .

---

1 **Algoritmo** *bwdSearch*(*i*, *d*)

**Input:** Índice a partir do qual a busca deve ser feita, excesso relativo buscado.

**Output:** Posição *j* onde ocorre o excesso *d* ou  $-1$  caso *d* não exista no intervalo definido.

```

2  dr  $\leftarrow$  0
3  l  $\leftarrow$   $\lfloor i / (b \cdot k) \rfloor$ 
4  v  $\leftarrow$  leafInTree(l)
5  reg  $\leftarrow$   $\lfloor ((i + 1) - (l \cdot b \cdot k)) / b \rfloor$ 
6  j  $\leftarrow$  bwdBlock(i, d, &dr) // Análogo ao algoritmo ??
7  if dr = d then
8    return j
9  reg  $\leftarrow$  reg - 1
10 if reg  $\geq$  0 then
11   j  $\leftarrow$  bwdRegistry((k · l + reg + 1) · b - 1, v, reg, l, d, &dr)
12   if dr = d then
13     return j

// Inicia o processo de subida na rmM-tree
14 while v  $\neq$  0 and bwdVerifySibling(&v, &dr, d)  $\neq$  -1 do
15   v  $\leftarrow$   $\lfloor (v - 1) / k \rfloor$ 
// Chegamos ao nó raiz, mas d não está em nenhum dos
// seus filhos
16 if v = 0 and reg = -1 then
17   return -1

// Desce nível a nível da árvore, até que v
// corresponda ao índice de um dos nós folhas
18 while v  $\leq$  numberNodes - numberLeaves do
19   for reg  $\leftarrow$  R[v].nReg - 1 to 0 do
20     if (dr - R[v][reg].e + R[v][reg].m  $\leq$  d  $\leq$  dr - R[v][reg].e + R[v][reg].M)
21       then
22         v  $\leftarrow$  (v · k) + 1 + reg
23         reg  $\leftarrow$  R[v].nReg - 1
24         break
25       else
26         dr  $\leftarrow$  dr - R[v][reg].e

26 l  $\leftarrow$  numLeaf(v)
27 if d = dr then
28   return (l · k · b) + (reg · b) - 1
29 j  $\leftarrow$  bwdRegistry((l · k · b) + ((reg + 1) · b) - 1, v, reg, l, d, &dr)
30 if dr = d then return j;
31 else return -1 ;

```

---

**Tabela 3.2:** Operações suportadas pela rmM-tree binária e rmM-tree-kária

<b>Operação</b>	<b>rmM-tree binária</b>	<b>rmM-tree k-ária</b>
fwdSearch(i,d)	✓	✓
bwdSearch(i,d)	✓	✓
minExcess(i,j) / maxExcess(i,j)	✓	✗
minCount(i,j)	✓	✗
minSelectExcess(i,j,t)	✓	✗
enclose(i)	✓	✓
rmq(i,j) / rMq(i,j)	✓	✗
rank <sub>1</sub> (i) / rank <sub>0</sub> (i)	✓	✓
select <sub>1</sub> (i) / select <sub>0</sub> (i)	✓	✓
preRank(i)/postRank(i)	✓	✓
preSelect(i)/postSelect(i)	✓	✓
isLeaf(i)	✓	✓
isAncestor(i,j)	✓	✓
depth(i)	✓	✓
parent(i)	✓	✓
firstChild(i) / lastChild(i)	✓	✓
child(i,t)	✓	✗
nextSibling(i) / prevSibling(i)	✓	✓
subtreeSize(i)	✓	✓
levelAncestor(i,d)	✓	✓
levelNext(i) / levelPrev(i)	✓	✓
levelLeftMost(d) / levelRightMost(d)	✓	✓
lca(i,j)	✓	✗
deepestNode(i)	✓	✗
degree(i)	✓	✗
childRank(i)	✓	✗
leafRank(i)	✓	✓
leafSelect(i)	✓	✓
leftMostLeaf(i)	✓	✓
rightMostLeaf(i)	✓	✓



## 4

### Resultados Experimentais

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

#### 4.1 Base de dados

Afim de realizar testes que comprovem o desempenho das estruturas implementadas na prática, usamos em nossos testes quatro conjuntos de dados do mundo real, estes conjuntos são apresentados em ?. A Tabela ?? 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, e por fim, a quarta coluna exibe a quantidade de nós da árvore representada pela sequência de parênteses balanceados.

**Tabela 4.1:** Conjunto de dados usados nos testes experimentais

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

Conforme nos mostra o site, o conjunto *Complete tree* representa uma árvore binária completa de profundidade igual à 30, ao passo que as sequências 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 (?).

## 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 possui 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

### 4.3.1 Decisões de projeto

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 tree.

Para realizarmos uma comparação justa entre a estrutura proposta por ? e a estrutura proposta neste trabalho, implementamos a rmM-tree no seu formato binário seguindo a descrição de ?. Após a implementação e validação dessa estrutura em seu formato binário, iniciamos a construção e validação da rmM-tree k-ária. Ambas as estruturas estão disponíveis no repositório [github.com/DanyelleAngelo/rmm-tree](https://github.com/DanyelleAngelo/rmm-tree), usando a linguagem de programação C++.

Durante os testes de desempenho (descritos com mais detalhes na seção ??), fizemos algumas adaptações na nossa proposta, assim implementamos duas versões da rmM-tree k-ária, estas se diferem unicamente pelo processo de subida na árvore. A primeira versão (adotada neste trabalho), segue a abordagem de verificação dos irmãos à direita (ou à esquerda, dependendo do sentido da busca) de um nó  $v$ , durante o percurso em árvore, como descrito por ?, para esta versão, o pior caso é observado durante uma busca iniciada em um nó  $v$ , com o excesso desejado não estando compreendido entre os registros dos irmãos de  $v$ , e  $v$  sendo um dos primeiros filhos

de seu nó pai. Buscando evitar este caso, construímos uma versão alternativa da rmM-tree k-ária, que não faz uso da técnica de visitação dos nós vizinhos de  $v$ , nesta versão ao terminarmos de inspecionar os registros de um nó  $v$ , avançamos na rmM-tree através do seu nó pai, verificando os registros deste nó, e inspecionando os demais filhos deste, apenas se a resposta estiver compreendida entre os registro de  $v$ .

Outra importante decisão no projeto das nossas estruturas e que melhorou significativamente o desempenho das mesmas, foi a implementação da função *bitsread* como é exposto na seção ???. Inicialmente essa função lia de modo iterativo  $w - 1$  bits a partir de um índice  $i$ , e realizava então uma pesquisa na tabela  $C$  pelo elemento correspondente aos bits lidos, obtendo assim os valores de excesso necessários para a computação de diversas operações. Essa leitura iterativa acrescentava um tempo considerável às nossas operações. Substituímos então esse processo iterativo, por um processo usando operações de deslocamento de bits, inserimos também em nosso projeto o uso de uma tabela de reversão, para que pudessemos realizar a leitura dos bits usando como critério de início, o bit mais significativo da palavra.

#### 4.3.2 Testes unitários

Objetivando garantir a asserção das respostas retornadas pela estrutura da rmM-tree binária e k-ária, realizamos a cada implementação – e alteração – testes unitários usando o framework Google Tests.

No início do desenvolvimento de cada estrutura 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), disponibilizada por ?. Para algumas operações foi necessário computar o valor de retorno 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 de validação da nossa estrutura.

Após os primeiros testes de unidade, expandimos o nosso escopo de testes. Utilizando a sequência de parênteses *Wikipedia*, geramos diversas consultas aleatórias afim de testar as operações da estrutura k-ária frente a estrutura binária.

No caso das operações que requeriam que o parâmetro de entrada correspondesse à um parênteses de abertura "(" ou de fechamento ")", foram pré-processados e selecionado de modo aleatório os índices que atendiam a especificação, afim de realizarmos testes com entradas válidas.

#### 4.3.3 Testes de Desempenho

Após a construção de cada estrutura e tendo concluído os testes de validação, iniciamos os testes de desempenho, para tanto foi usado o framework Google Benchmark. Durante esses testes, foram introduzidas as melhorias citadas na seção de *Decisões de projeto*, sendo que à cada nova alteração, realizávamos novamente, todos os testes unitários, e testes de desempenho,

afim de garantir a aserção dos nossos resultados.

Para os testes de desempenho, foram pré-computados diversos vetores pseudo-aleatórios, usando diferentes sementes (afim de manter uma variedade maior dos parâmetros usados), para diferentes operações. Cada um desses vetores tinham como entrada índices dos conjuntos de dados usados nos testes, que caracterizam parâmetros válidos para as operações sobre as rmM-tree implementadas.

Para cada operação foi usado cerca de 3.000.000 de dados, os resultados de desempenho das mesmas é mostrado na próxima seção.

#### 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 (barra verde), e
- rmM-tree 16-ária (barra vermelha).

O tamanho de bloco usado em todos os testes foi definido como 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 nossos gráficos se refere ao tempo médio para 3.000.000 consultas sobre as diferentes versões da rmM-tree, 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 de operações de acesso ao vetor que representa a árvore de entrada.

Como pode ser observado nos gráficos, a rmM-tree binária obteve melhor desempenho para todas as operações, isso está relacionado ao fato de que a rmM-tree binária, possui uma série de otimizações a nível de código, que não são implementadas na rmM-tree k-ária. Por exemplo, para a rmM-tree binária, o percurso em árvore é interrompido sempre que atingimos o último nó de um nível (ou primeiro de um nível, para o caso da operação *bwdSearch*).

Para as aridades da rmM-tree k-ária analisadas, não foi possível detectar um padrão para os diferentes conjuntos de dados. No geral, o desempenho das estruturas *k-árias* variam entre si, para cada operação em unidades de *nanossegundos*. Com base no conteúdo abordado neste trabalho, acredita-se que mediante à implementação de otimizações no código da rmM-tree k-ária, poderemos atingir o equilíbrio entre a quantidade de instruções executadas, e acessos a memória principal, melhorando assim o desempenho do sistema.



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

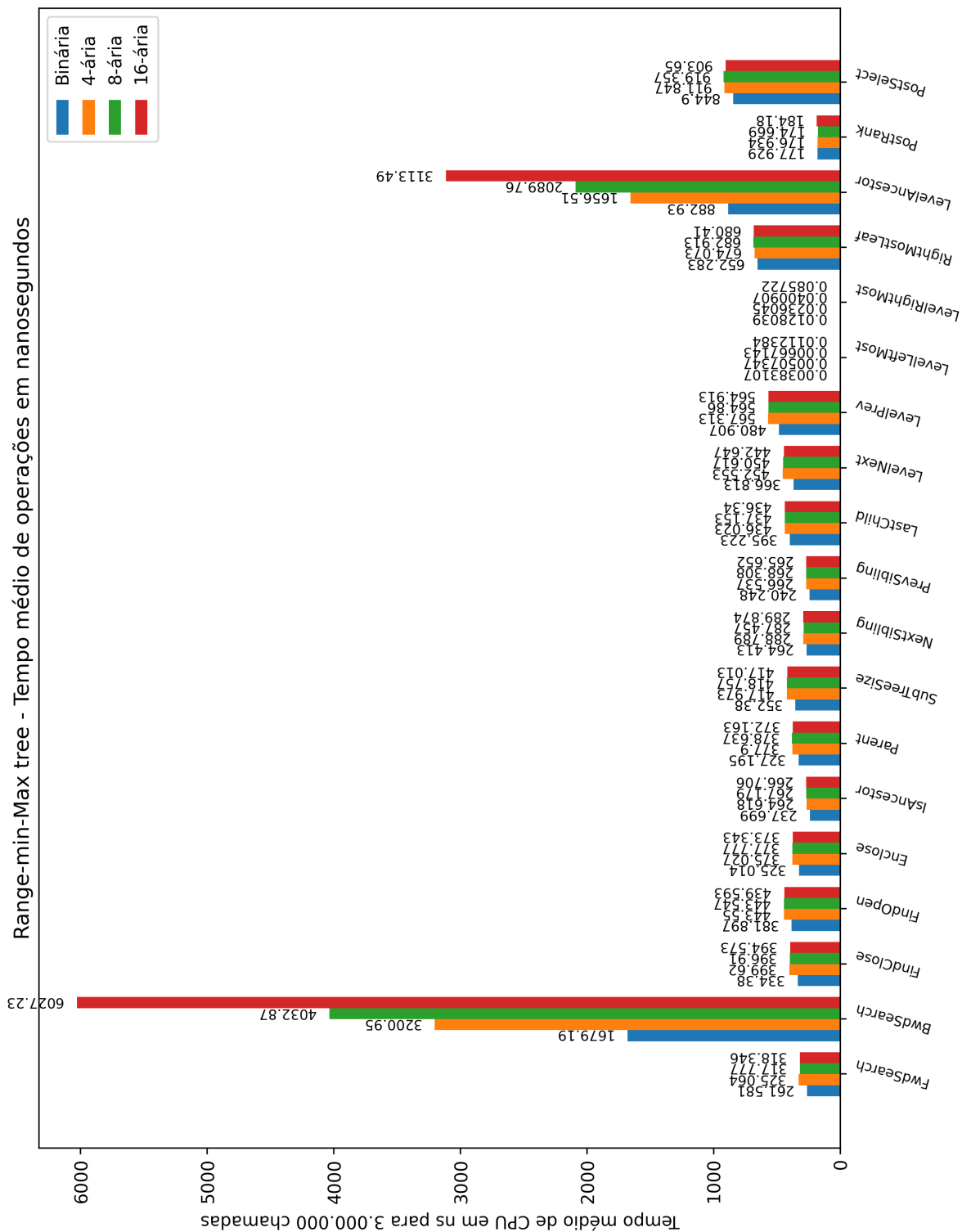


Figura 4.2: Tempo médio de operações sobre o conjunto DNA

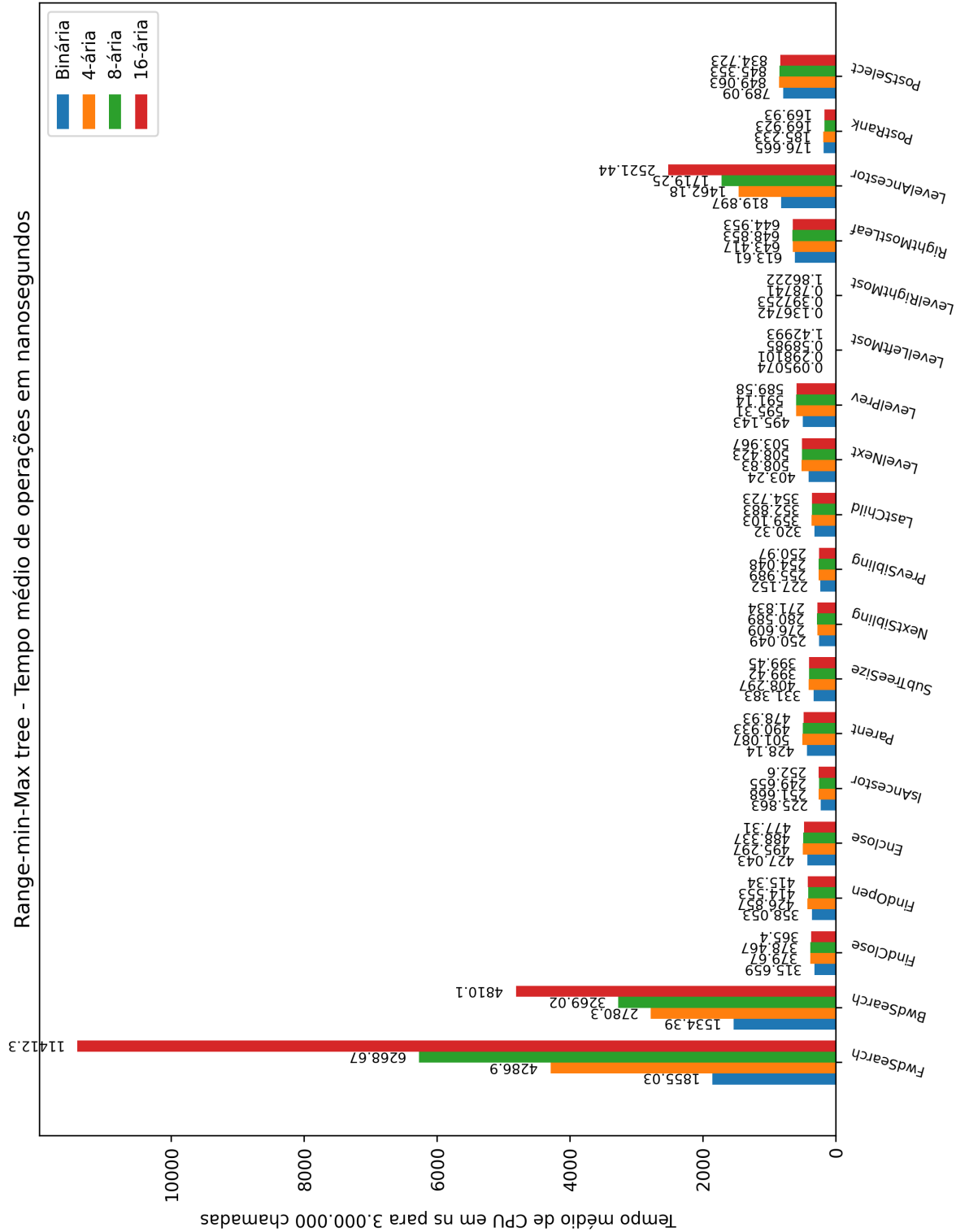


Figura 4.3: Tempo médio de operações sobre o conjunto Proteins

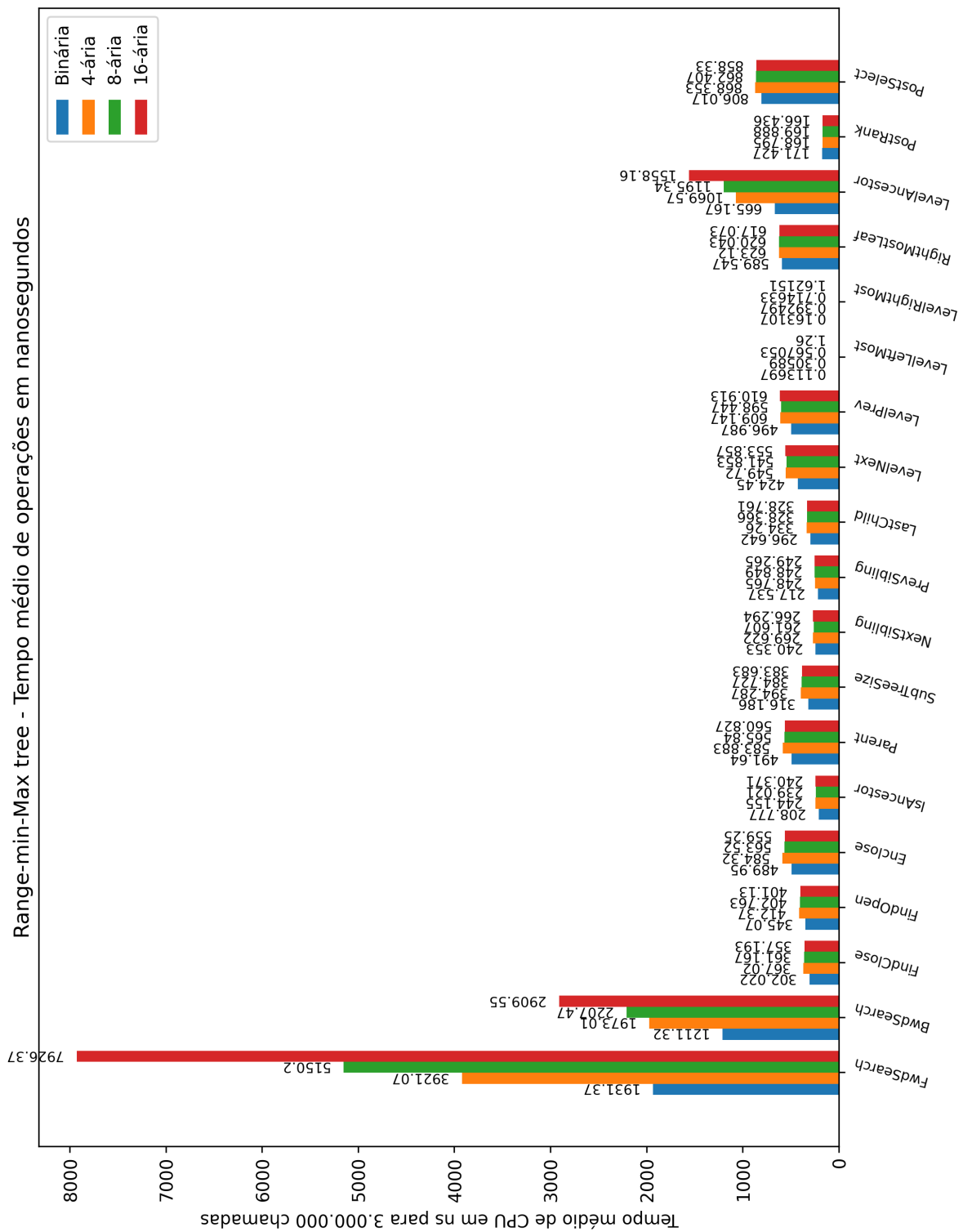
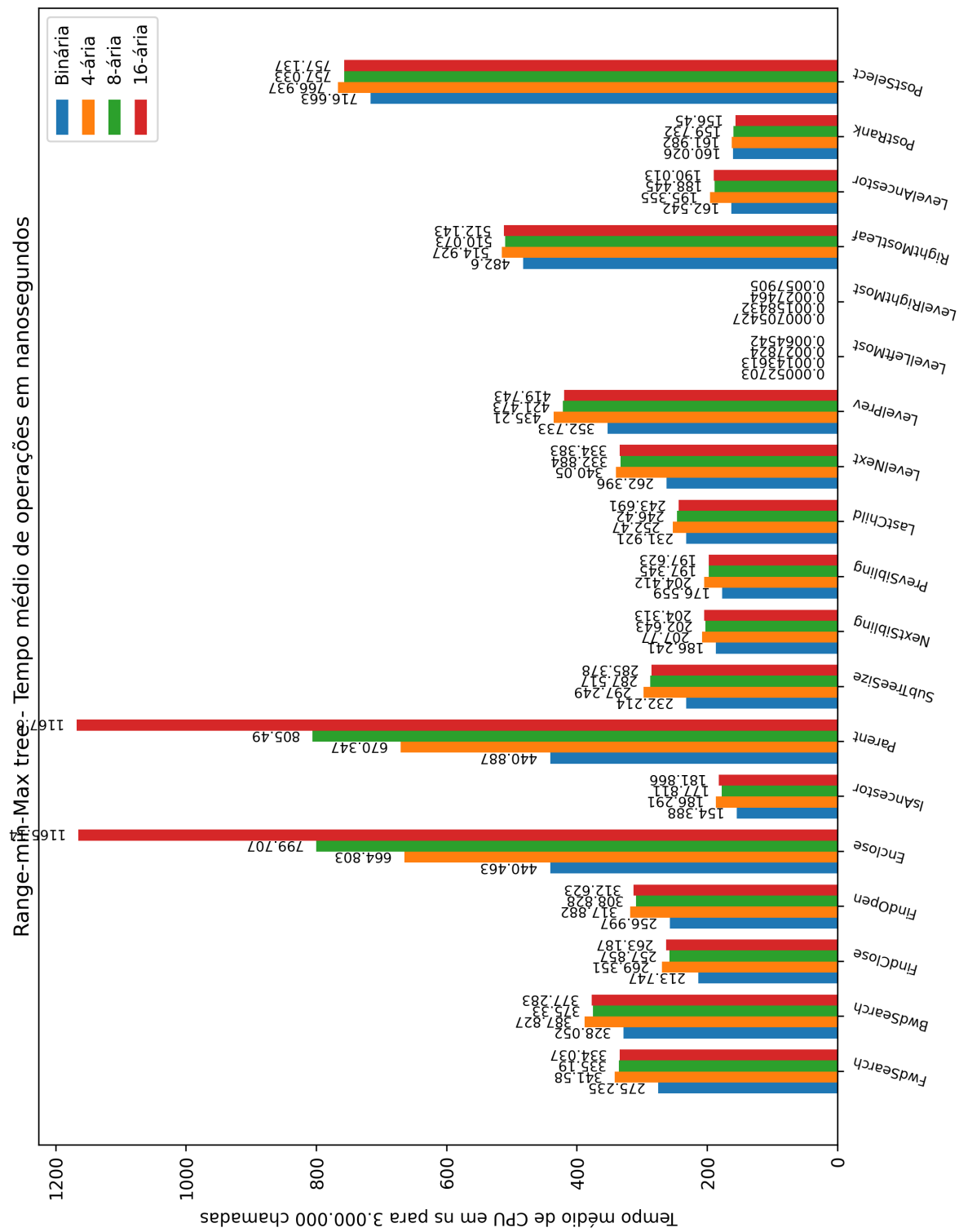


Figura 4.4: Tempo médio de operações sobre o conjunto Wikipedia





## 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 alta latência dessas memórias.

As estruturas de dados sucintas, possibilitam a representação e operação sobre os seus objetos, usando espaço e tempo reduzido. Assim um dos objetivos deste trabalho era compreender e enfatizar a importância do estudo dessas estruturas. Entretanto, ao operar na memória principal, temos o surgimento de um novo gargalo de desempenho, dessa vez entre cache e memória principal, como mostram [1], faz-se necessário então, buscarmos maneiras de melhor aproveitar a estrutura da memória cache. Uma das maneiras de fazer isso é maximizando o uso de uma linha de cache, que para aplicações que usam árvores, pode ser feito através da maximização do fator de ramificação destas estruturas.

A rmM-tree é uma estrutura de dados sucinta, que permite a navegação de forma eficiente em árvore, através de valores de excesso máximos e mínimos em intervalos, porém a mesma é construída na forma de árvore binária, uma estrutura que possui baixo fator de ramificação, e portanto baixo aproveitamento da linha de cache, nosso objetivo neste trabalho portanto era aumentar o fator de ramificação dessa estrutura, visando uma diminuição do número de faltas de cache, como visto em [2], o que reduziria significativamente o tempo gasto para realizar operações diversas.

Implementamos e analisamos, para tanto, uma versão da rmM-tree de [1], e 3 versões de uma rmM-tree k-ária, compreendendo 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. Comparando a rmM-tree binária, frente às diferentes versões da rmM-tree k-ária, a primeira teve melhor desempenho em todas as operações. Em relação à rmM-tree k-ária, não foi possível detectar um padrão de comportamento para os diferentes conjuntos de dados usados nos testes.

Para trabalhos futuros temos como objetivo reduzir o tempo das operações, através da

otimização da implementação da rmM-tree k-ária. É necessário 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.