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

Estruturas de Dados Sucintas

Maximilian Cabrajac Göritz

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisora: Profª. Drª. Cristina Gomes Fernandes

São Paulo
2024

Resumo

Maximilian Cabrajac Göritz. **Estruturas de Dados Sucintas**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

Com o surgimento de campos de estudo e aplicação cuja quantidade de dados a serem analisados é massiva, como ferramentas de busca e sequenciamento genético, tornaram-se necessários avanços teóricos em busca de ferramentas que permitissem tratar essa imensidão de informação. Entre essas ferramentas estão as estruturas de dados sucintas.

Essas estruturas são categorizadas pelo seu consumo sublinear de espaço adicional, ou seja, se a informação na estrutura requer n bits para ser armazenada, para que a estrutura seja sucinta, o número de bits para representá-la deve ser $n + o(n)$.

Neste trabalho, são apresentadas duas representações sucintas para árvores binárias: uma mais simples e uma mais complexa. As duas permitem navegar pela árvore com custo constante no modelo computacional adotado, mas somente a segunda permite encontrar o tamanho da subárvore enraizada em um nó qualquer com custo constante, além de ser possível estendê-la para dar suporte a algumas operações sobre suas folhas.

Palavras-chave: Estruturas de dados sucintas. Árvores binárias sucintas. Rank e Select. Sequências balanceadas de parênteses.

Abstract

Maximilian Cabrajac Göritz. **Sucinct Data Structures**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, Brazil, 2024.

With the increase in research and application areas where massive amounts of data have to be processed, such as search engines and genetic sequencing, theoretic advancements on methods to handle such information became necessary. Among these methods are succinct data structures.

These structures are characterized by their use of sublinear additional space, that is, if the information on a structure requires n bits to be stored, for the structure to be succinct, the number of bits to represent it must be $n + o(n)$.

In this work, two representations for binary trees are shown: a simple and a more complex one. Both allow efficient navigation on the tree, but only the second allows one to efficiently get the size of the subtree rooted on some node. Additionally, the second one can be extended to perform some operations on leaves.

Keywords: Succinct data structures. Binary trees. Rank and Select. Balanced sequences of parentheses.

Lista de figuras

Lista de programas

Sumário

Introdução

Estruturas de Dados Sucintas

Um dos métodos utilizados para economizar espaço ao armazenar grandes volumes de dados é a compressão. Independente do algoritmo de compressão utilizado, esse método se baseia em transformações custosas entre dois estados de uma mesma informação: o estado bruto e o estado comprimido. Enquanto o estado bruto consome bastante espaço e permite a análise direta do conteúdo, o estado comprimido tenta economizar o máximo de espaço possível em detrimento da compreensibilidade do que é armazenado.

Embora técnicas de compressão sejam muito úteis, há situações onde é fundamental que a informação consuma pouco espaço e, simultaneamente, seja compreensível, o que requer que outros métodos sejam desenvolvidos.

No início da busca por métodos de economia de espaço ao armazenar algum objeto, um primeiro passo natural é buscar por delimitações inferiores para o espaço necessário para representá-lo. Um resultado simples da teoria da informação estabelece que qualquer representação dos elementos de um conjunto S tem pelo menos $\lg |S|$ bits.

Munido desse fato, Jacobson [Jac88] define três categorias de estruturas de dados, isto é, representações de algum conjunto de objetos que permitem uma coleção de operações. Seja n o limite teórico para representações de elementos desse conjunto. Estruturas de dados:

- Canônicas consomem $n + \mathrm{O}(1)$ bits.
- Assintoticamente ótimas consomem $n(1 + \mathrm{o}(1)) = n + \mathrm{o}(n)$ bits.
- Lineares consomem $\mathrm{O}(n)$ bits.

Com a popularização desse campo de estudo, essas categorias tiveram suas nomenclaturas simplificadas, respectivamente, para estruturas de dados implícitas, sucintas e compactas. Entre essas, as estruturas de dados sucintas, foco desse trabalho, se mostram muito interessantes, uma vez que a razão entre o consumo de espaço da estrutura e o limite teórico para essa tende a 1 com o crescimento do objeto representado.

Modelo Computacional

Um modelo básico de computação é o RAM [Emd91]. Esse é composto por um espaço finito de memória que contém o programa, um ou mais registradores acumuladores e

infinitos registradores de memória, onde ambos os tipos de registradores têm capacidade infinita, mas em qualquer estado alcançável representam um valor finito. Com isso, suas primitivas permitem:

- Controle de fluxo, possivelmente baseado no conteúdo de registradores acumuladores.
- Entrada e saída.
- Transporte de dados bidirecional entre registradores acumuladores e registradores de memória.
- Operações aritméticas entre acumuladores, inicialmente, soma e subtração.

A adição de alguns prefixos à sigla do modelo refere-se a algumas variantes do modelo RAM [Emd91]:

- S: Deixa de permitir soma e subtração e passa a permitir incremento e decremento como operações aritméticas.
- M: Adiciona multiplicação e divisão ao repertório de operações aritméticas.
- B: Permite operações bit a bit, como *and*, *or* e *shifts* entre acumuladores.

A partir desses prefixos, pode-se formar, por exemplo, o modelo MBRAM, que se diferencia do modelo RAM por possuir primitivas para multiplicação, divisão e operações bit a bit.

O MBRAM *comum* serve de base para os dois modelos de computação utilizados com maior frequência no estudo sobre estruturas de dados sucintas: o MBRAM *de barramento estreito* e o MBRAM *de barramento largo* [Jac88; Cla97].

Ambos os modelos utilizados divergem do MBRAM comum ao limitarem a capacidade dos registradores acumuladores a $\Theta(\lg n)$ bits onde n é uma medida razoável do tamanho da instância do problema e substituem os registradores de memória por uma fita de memória contínua, infinita e que permite acesso aleatório. Os modelos com diferentes barramentos divergem entre si ao lidar com o transporte de dados entre a fita e seus registradores: no modelo de barramento estreito somente um bit pode ser transportado por operação, já no modelo de barramento largo, o conteúdo de uma região contínua, seja ela pertencente à fita ou a um registrador, de tamanho menor ou igual ao de um registrador, pode ser transportado em uma única operação.

Note que um algoritmo qualquer de custo $O(f(n))$ no modelo MBRAM de barramento largo pode ser descrito como um algoritmo de custo $O(f(n) \lg n)$ no modelo MBRAM de barramento estreito, uma vez que a operação de transporte do primeiro pode ser simulada no segundo por meio de $O(\lg n)$ operações. Contudo, o inverso é falso: devido à preocupação do MBRAM de barramento largo com a localidade da informação, existem algoritmos com custo $O(f(n) \lg n)$ no MBRAM de barramento estreito que não podem ser descritos como um algoritmo de custo $O(f(n))$ no MBRAM de barramento largo.

No decorrer do texto, o modelo utilizado é o MBRAM de barramento largo, devido à sua simplicidade e semelhança aos computadores da atualidade. Nos trechos de pseudocódigo as operações de transporte de memória serão representadas de duas formas. Quando o

acesso a uma determinada região da fita é feito de forma padronizada, ou seja, todas as leituras e escritas têm o mesmo tamanho, a região de memória será representada na forma de um vetor ou matriz comum e o tamanho de cada uma de suas células será explicitado textualmente. Para regiões acessadas de forma não padronizada, as quais chamaremos de *vetor de bits*, o acesso será feito por meio das primitivas **LerDeVetorDeBits** e **EscreverEmVetorDeBits**. Ambas recebem um vetor de bits, uma posição relativa ao início desse e o número de bits da região de memória a ser acessada, além disso, a segunda recebe como último argumento o valor a ser escrito.

Por razões de concisão, a exponenciação com base 2, ou seja, algo como 2^x, e o acesso direto a posições de vetores de bits, cujo resultado esperado é o booleano correspondente ao bit acessado, serão utilizados em trechos de pseudocódigo. Note que ambos podem ser calculados com custo constante no modelo, respectivamente, por meio de shifts e leituras de bits únicos.

Capítulo 1

Rank e Select

No campo de estudo sobre estruturas de dados sucintas, onde o componente unitário é o bit em vez de inteiros ou objetos, duas operações são recorrentemente utilizadas na construção de estruturas complexas. Dado que v é um vetor de n bits, essas são:

- Rank(i) é o número de bits 1 do início até, incluindo, a i-ésima posição de v.
- Select(i) é o índice do i-ésimo bit 1 de v.

Neste capítulo, são apresentadas as estruturas sucintas para calcular as operações Rank e Select com custo constante, propostas por Jacobson [Jac88] e Clark [Cla97], respectivamente.

1.1 Rank

A solução sucinta para Rank descrita por Jacobson [Jac88] pré-processa o vetor de bits construindo três tabelas que possibilitam obter o Rank de uma posição eficientemente.

No início do pré-processamento é fixado o valor $L = \left\lceil \frac{\lg n}{2} \right\rceil$, que dita o formato das tabelas e outras particularidades da estrutura. Depois disso, divide-se o vetor v em trechos grandes, de comprimento L^2, e armazena-se o Rank absoluto da posição que antecede o início de cada trecho em uma tabela denominada `rank_absoluto`, indexada pelo índice do trecho.

Em sequência, estes trechos são subdivididos em subtrechos pequenos, de comprimento L, e os Ranks relativos ao trecho das posições que antecedem o início dos subtrechos são armazenados em uma tabela denominada `rank_relativo`, indexada pelo índice do trecho e pelo índice do subtrecho no trecho.

Por fim, é criada uma tabela, chamada de `rank_local`, na qual é armazenado, para cada par (c, i), onde c é o conteúdo de um subtrecho e i uma posição em c, o Rank de i em c.

Dessa forma o Rank de uma posição arbitrária do vetor pode ser recuperado por meio da soma de um valor de cada tabela. Veja a Figura 1.1.

Tabelas que correlacionam todos os números de $\left\lceil \frac{\lg n}{2} \right\rceil$ bits a informações locais, como `rank_local`, são muito utilizadas na construção de estruturas de dados sucintas. Elas são usualmente chamadas de *tabelas de consulta*.

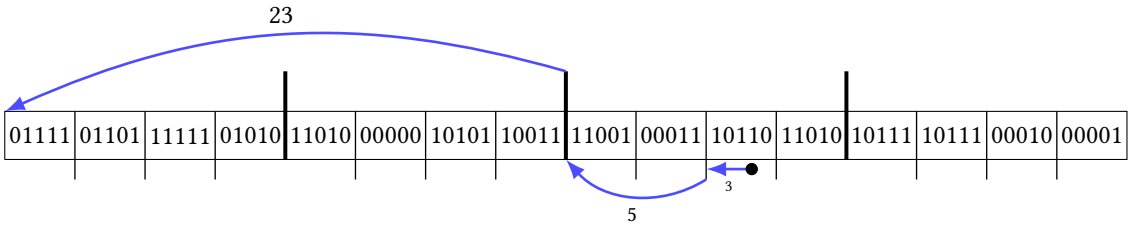

Figura 1.1: *Exemplo de* Rank *com resultado* $23 + 5 + 3 = 31$.

1.1.1 Pré-processamento

Para construir as tabelas `rank_absoluto` e `rank_relativo`, basta iterar uma vez pelo vetor de bits atualizando essas quando necessário, o que tem custo $\Theta(n)$ no modelo. Para fins de simplicidade, os trechos e subtrechos são indexados a partir de 0. Veja o Programa 1.1.

Programa 1.1 Construção das tabelas `rank_absoluto` e `rank_relativo`

Recebe L, n e o vetor de bits $v[1..n]$.

```
1   rank_absoluto[0] ← 0
2   rank_relativo[0][0] ← 0
3   ind_trecho ← 0
4   ind_subtrecho ← 0
5   c ← 0
6   para i ← 1 até n
7      se v[i] = 1
8         c ← c + 1

9      se i mod L² = 0
10        ind_trecho ← ind_trecho + 1
11        ind_subtrecho ← 0
12        rank_absoluto[ind_trecho] ← c
13        rank_relativo[ind_trecho][0] ← 0

14     se i mod L = 0
15        ind_subtrecho ← ind_subtrecho + 1
16        c_relativo ← c − rank_absoluto[ind_trecho]
17        rank_relativo[ind_trecho][ind_subtrecho] ← c_relativo
```

Já para construir a tabela de consulta `rank_local` é necessário iterar por todas as entradas possíveis e catalogar as saídas pertinentes. Veja o Programa 1.2.

O código apresentado é correto e explicita a varredura de todos os possíveis conteúdos de um subtrecho, assim como a manipulação dos bits da variável `conteúdo` por meio de comparações e subtrações.

Programa 1.2 Construção da tabela `rank_local`

Recebe L.

```
1  para i ← 0 até 2^L − 1
2     conteúdo ← i
3     c ← 0
4     para j ← 1 até L
5        se conteúdo ≥ 2^(L−j)
6           c ← c + 1
7           conteúdo ← conteúdo − 2^(L−j)
8        rank_local[i][j] ← c
```

Como tabelas de consulta são muito comuns, nos programas à frente, quando apresentarmos a construção de outras tabelas de consultas, optamos por uma versão mais compacta, na qual mostramos apenas a construção de uma linha da tabela, ou seja, dado um valor para `conteúdo`, mostraremos apenas como é preenchida a linha da tabela de consulta correspondente a essa valoração. Para calcular essa linha, acessaremos os bits de `conteúdo` usando a notação matricial usual, ou seja, `conteúdo` será utilizado com um vetor de bits. Para armazenar os valores que compõem a linha construída, `conteúdo` será utilizado como índice da linha da tabela de consulta, omitindo uma chamada a **LerDeVetorDeBits** que copiaria todo o vetor de bits para uma variável numérica. Desta forma, o Programa 1.2 seria simplificado para o Programa 1.3.

Programa 1.3 Construção de uma linha da tabela `rank_local`

Recebe L e um vetor de bits `conteúdo`$[1..L]$.

```
1  c ← 0
2  para j ← 1 até L
3     se conteúdo[j]
4        c ← c + 1
5     rank_local[conteúdo][j] ← c
```

Passando à analise de complexidade, tem-se que, uma vez que existem 2^L, ou seja, $\Theta(\sqrt{n})$ números representáveis com L bits e a construção da linha de `rank_local` referente a cada um desses tem custo $O(\lg n)$, sua construção completa tem custo $\Theta(\sqrt{n} \lg n)$.

Portanto, o custo de construção de `rank_local` é dominado pelo custo da construção de `rank_absoluto` e `rank_relativo`, ou seja, o custo de construir as três estruturas é linear no comprimento do vetor de bits.

A constante L e as três tabelas construídas no pré-processamento serão utilizadas no algoritmo que executa a consulta sem serem passados explicitamente como parâmetros no correspondente algoritmo.

1.1.2 Utilização

Para obter o Rank de uma posição i, primeiramente é necessário calcular o índice do trecho e do subtrecho de v que contém a posição i. Em sequência, lê-se de v o conteúdo

do subtrecho com a rotina **LerSubtrecho** e calcula-se o índice da posição i dentro do subtrecho. Especificamente, a rotina **LerSubtrecho** recebe dois índices i e j, onde i é o índice de um trecho de v e j é o índice de um de seus subtrechos, e devolve uma cópia do subtrecho em questão, eventualmente acrescida de zeros à direita para completar o comprimento L. Como cada subtrecho tem comprimento $L = O(\lg n)$, a rotina **LerSubtrecho** faz somente uma chamada à primitiva **LerDeVetorDeBits**, ou seja, **LerSubtrecho** tem custo constante.

Com essas informações, utiliza-se as tabelas criadas no pré-processamento, para calcular o Rank requisitado, conforme o Programa 1.4. Esse cálculo tem custo constante no modelo.

Programa 1.4 Cálculo do Rank

Recebe um índice i de v, $1 \leq i \leq n$.

```
1  função Rank(i)
2      ind_trecho ← i/L²
3      ind_subtrecho ← (i mod L²)/L
4      conteúdo_subtrecho ← LerSubtrecho(ind_trecho, ind_subtrecho)
5      posição_subtrecho ← i mod L
6      devolva rank_absoluto[ind_trecho] +
7          rank_relativo[ind_trecho][ind_subtrecho] +
8          rank_local[conteúdo_subtrecho][posição_subtrecho]
```

1.1.3 Análise de espaço

Para fazer a análise de espaço das estruturas utilizadas para computar o Rank é necessário verificar a quantidade de entradas em cada tabela e o número de bits consumidos por cada uma delas.

A tabela `rank_absoluto` tem $\left\lceil \frac{n}{L^2} \right\rceil$, ou seja, $\Theta\left(\frac{n}{\lg^2 n}\right)$ entradas e cada uma dessas ocupa $\lceil \lg n \rceil$ bits, portanto `rank_absoluto` requer $\Theta\left(\frac{n}{\lg n}\right)$ bits.

A tabela `rank_relativo` contém $\left\lceil \frac{n}{L^2} \right\rceil \cdot L = \Theta\left(\frac{n}{L}\right)$ entradas. Por outro lado, não é necessário que elas utilizem $\lceil \lg n \rceil$ bits, posto que comportam somente o Rank dentro dos trechos. Desta forma, o maior número a ser representado se torna L^2, ocupando $\lceil 2 \lg L \rceil$, ou seja, $\Theta(\lg \lg n)$ bits. Logo, o consumo total de espaço para `rank_relativo` é $\Theta\left(\frac{n \lg \lg n}{\lg n}\right)$.

Já a tabela `rank_local` mantém todas as $L = \Theta(\lg n)$ respostas para Rank para cada um dos $2^L = \Theta(\sqrt{n})$ possíveis números representáveis com tal quantidade de bits. Cada uma dessas respostas ocupa somente $\lg L = \Theta(\lg \lg n)$ bits. Em virtude disso, `rank_local` consome espaço $\Theta(\sqrt{n} \lg n \lg \lg n)$.

Como o consumo de espaço das três estruturas é assintoticamente menor que n, ou seja, $o(n)$, a estrutura criada para permitir o cálculo eficiente do Rank é sucinta.

1.2 Select

A solução apresentada por Clark [Cla97] para a operação Select consiste em cinco tabelas. Para computar a operação eficientemente, são utilizadas até três dessas tabelas.

No início do pré-processamento são calculados dois valores $L = \left\lceil \frac{\lg n}{2} \right\rceil$ e $S = \left\lceil \frac{\lg \lg n}{2} \right\rceil$. Com isso, o pré-processamento procede construindo as tabelas. A primeira tabela, denominada `separadores1`, contém os índices absolutos de cada iLS-ésimo bit 1 de v para $i \geq 1$.

Esses bits 1 dividem v em setores de tamanho variável. Denote como *limiar de densidade* um valor d que categoriza os setores em dois tipos: aqueles cujo comprimento é menor ou igual a d e aqueles cujo comprimento é estritamente maior que d. Diz-se que estes setores foram categorizados pelo limiar d, respectivamente, como *densos* e *esparsos*.

Os setores gerados por `separadores1` são categorizados pelo limiar $(LS)^2$. Cada setor esparso tem todos os seus bits 1 catalogados em uma das linhas da matriz denominada `esparsos1`. Já cada setor denso tem os índices relativos de cada iS^2-ésimo de seus bits 1 para $i \geq 1$ mantidos em uma das linhas da matriz `separadores2`. Analogamente, tais bits 1 de um setor denso o dividem em subsetores de tamanho variável.

Neste segundo nível, utiliza-se como limiar de densidade S^4. Se um subsetor é esparso, os índices relativos de todos os seus bits 1 são armazenados em uma linha da terceira matriz, denominada de `esparsos2`.

Por fim, se um subsetor é denso, a posição de cada um de seus bits 1 pode ser extraída da tabela de consulta `densos`, que mantém o número de bits 1 em cada trecho de comprimento L possível, além da posição de cada um deles dentro do trecho. Especificamente, para encontrar a posição de um bit 1 de um subsetor denso de v, lê-se um trecho de comprimento L do subsetor e por meio do número de bits 1 de tal trecho, guardado em `densos`, pode-se saber se o bit 1 procurado está nesse trecho ou num trecho seguinte do subsetor. Encontrado o trecho do subsetor denso que contém o bit 1 requisitado, pode-se utilizar das outras informações de `densos` para encontrar sua posição. Isso será detalhado adiante.

1.2.1 Pré-processamento

A construção das estruturas descritas para a operação Select segue a sequência do Programa 1.5, no qual as tarefas envolvidas em subrotinas são detalhadas adiante.

Desta forma, descrever e analisar as sub-rotinas isoladamente se torna mais simples. Em um primeiro momento, na linha 3, é construído o vetor `separadores1`. Para construir esse vetor basta iterar sobre v catalogando a posição de cada iLS-ésimo bit 1 para $i \geq 1$. Isto tem custo $\Theta(n)$ no modelo, conforme o Programa 1.6.

A partir dos dados de `separadores1`, é simples calcular o início, o fim e por consequência o comprimento de um setor, dado seu índice, o que é feito, respectivamente, pelas funções **InícioDoSetor**, **FimDoSetor** e **ComprimentoDoSetor** no Programa 1.7. Todas essas operações têm custo constante.

Partindo para a construção de `esparsos1`, encontra-se um pequeno problema. Como

Programa 1.5 Sequência para construção das estruturas para o SELECT

```
1   L ← ⌈lg n/2⌉
2   S ← ⌈lg lg n⌉
3   C ← ConstroiSeparadores1()                    ▷ C: índice do último setor
4   i ← 0
5   enquanto i ≤ C
6       se ComprimentoDoSetor(i) > (LS)²
7           ConstroiEsparsos1ParaSetor(i)
8       senão
9           ConstroiSeparadores2ParaSetor(i)
10          j ← 0
11          enquanto j ≤ L/S
12              se ComprimentoDoSubsetor(i, j) > S⁴
13                  ConstroiEsparsos2ParaSubsetor(i, j)
14              j ← j + 1
15      i ← i + 1
16  ConstroiDensos()
```

Programa 1.6 Construção do vetor `separadores1`

```
1   função ConstroiSeparadores1()
2       s ← LS
3       c ← 0
4       para i ← 1 até n
5           se v[i]
6               c ← c + 1
7               se c mod s = 0
8                   separadores1[c / s] ← i
9       separadores1[0] ← 0                       ▷ valor sentinela
10      separadores1[1 + c / s] ← n + 1           ▷ valor sentinela
11      devolva c / s
```

cada entrada da matriz `esparsos1` deve poder armazenar $\lceil \lg n \rceil$ bits, capacidade necessária, por exemplo, quando o vetor v é composto por somente um setor esparso, é necessária cautela ao determinar seu tamanho.

Note que o número máximo de setores esparsos é $\left\lceil \frac{n}{(LS)^2} \right\rceil$, pois cada setor esparso tem comprimento pelo menos $(LS)^2$. A matriz `esparsos1` comportará esse número de linhas.

Mas em que linha de `esparsos1` se guarda a informação referente a um determinado setor esparso? O índice i do setor varia de 0 até o C, obtido na linha 3 do Programa 1.5. Logo o índice i não é uma opção, pois C pode ser muito maior que $\left\lceil \frac{n}{(LS)^2} \right\rceil$. Também não se pode calcular rapidamente (com custo constante) quantos setores esparsos precedem um dado setor, ou seja, utilizar o índice do setor dentre os esparsos é inviável.

Contudo, como citado anteriormente, todos os setores esparsos têm comprimento maior ou igual a $(LS)^2$. De posse dessa informação, o índice do primeiro bit de um setor

Programa 1.7 Funções auxiliares para setores

```
1  função InícioDoSetor(i)
2      devolva separadores1[i]

3  função FimDoSetor(i)
4      devolva InícioDoSetor(i + 1) − 1

5  função ComprimentoDoSetor(i)
6      devolva FimDoSetor(i) − InícioDoSetor(i) + 1
```

esparso dividido por $(LS)^2$ o identifica univocamente entre os outros setores esparsos. Note que, para um possível índice de setor esparso, não é garantido que exista um setor esparso correspondente, já que o trecho em que esse setor se encontraria pode ser coberto por setores densos.

Agora, para construir a linha de `esparsos1` referente a um dado setor esparso, é necessário iterar sobre este, catalogando as posições de todos os bits 1 do setor, o que tem custo linear no comprimento do setor. Veja o Programa 1.8.

Programa 1.8 Construção de `esparsos1` referente a um setor

```
1  função ConstroiEsparsos1ParaSetor(i)
2      id ← InícioDoSetor(i)/(LS)²
3      c ← 0
4      para j ← InícioDoSetor(i) até FimDoSetor(i)
5          se v[j]
6              esparsos1[id][c] ← j
7              c ← c + 1
```

De forma similar ao feito na construção de `separadores1`, durante a construção da linha de `separadores2` referente a um setor denso, são registradas as posições dos bits 1 em intervalos periódicos, a fim de dividir o setor em subsetores. Para isso, é necessário iterar sobre o setor, o que tem custo linear no comprimento do mesmo. Veja o Programa 1.9.

Programa 1.9 Construção de `separadores2` referente a um setor

```
1  função ConstroiSeparadores2ParaSetor(i)
2      s ← S²
3      c ← 0
4      para j ← InícioDoSetor(i) até FimDoSetor(i)
5          se v[j]
6              c ← c + 1
7              se c mod s = 0
8                  separadores2[i][c / s] ← j − InícioDoSetor(i)
9      separadores2[i][0] ← 0                          ▷ valor sentinela
10     para k ← 1 + c / s até L/S
11         separadores2[i][k] ← FimDoSetor(i)          ▷ valor sentinela
```

Vale notar que se um setor de índice i é esparso, a i-ésima linha de `separadores2` não será utilizada pelo programa.

Assim como **InícioDoSetor**, **FimDoSetor** e **ComprimentoDoSetor**, pode-se descrever suas versões relativas a subsetores, cada uma com custo constante. Veja o Programa 1.10.

Programa 1.10 Funções auxiliares para subsetores

```
1  função InícioDoSubsetor(i, j)
2      devolva InícioDoSetor(i) + separadores2[i][j]

3  função FimDoSubsetor(i, j)
4      devolva InícioDoSubsetor(i, j + 1) − 1

5  função ComprimentoDoSubsetor(i, j)
6      devolva FimDoSubsetor(i, j) − InícioDoSetor(i, j) + 1
```

Semelhante à construção de `esparsos1` para um setor, a construção da linha de `esparsos2` que corresponde a um subsetor o percorre armazenando as posições de todos os bits 1 que este compreende e utilizando para indexação a divisão do início do subsetor pelo limitante inferior do comprimento desse, que no caso é S^4. Desta forma, para um dado subsetor, a construção de `esparsos2` possui custo linear no comprimento desse. Veja o Programa 1.11.

Programa 1.11 Construção de `esparsos2` referente a um subsetor

```
1  função ConstroiEsparsos2ParaSubsetor(i, j)
2      id ← InícioDoSubsetor(i, j) / S⁴
3      c ← 0
4      para j ← InícioDoSubsetor(i, j) até FimDoSubsetor(i, j)
5          se v[j]
6              esparsos2[id][c] ← j − InícioDoSetor(i)
7              c ← c + 1
```

Para criar a tabela de consulta `densos`, itera-se sobre os bits de `conteúdo` armazenando as posições dos bits 1 além da quantidade dos mesmos, totalizando custo $O(\sqrt{n} \lg n)$. Veja o Programa 1.12.

Programa 1.12 Construção da tabela `densos`

```
1  função ConstróiDensos(conteúdo)
2  c ← 0
3  para i ← 1 até L
4      se conteúdo[i]
5          c ← c + 1
6          densos[conteúdo][c] ← i
7      densos[conteúdo][0] ← c
```

Note que, na 0-ésima posição da linha pertinente a cada um dos conteúdos possíveis, há a quantidade de bits 1 que esse possui. Desta forma, pode-se obter essa informação em somente um acesso.

Uma vez que todas as sub-rotinas que compõem a construção das estruturas utilizadas para a operação Select foram analisadas é possível analisar a construção em sua totalidade.

A construção de `esparsos2` para um subsetor esparso tem custo linear no comprimento desse. Desta forma, no pior caso, a construção de `esparsos2` para todos os subsetores esparsos que compõem um setor denso tem custo linear no comprimento do setor, assim como a construção da linha de `separadores2` para o mesmo setor. Com isso, a construção das estruturas referentes a setores densos possui custo linear no comprimento do setor.

Da mesma maneira, a construção de `esparsos1`, estrutura referente a setores esparsos, tem custo linear no comprimento do setor.

Com esses fatos, pode-se concluir que o custo de construção das estruturas relevantes a um setor, seja ele denso ou esparso, é linear em seu comprimento. A partir disso, a construção das estruturas referentes a todos os setores tem custo $\Theta(n)$, assim como a construção de `separadores1`. Assim, conclui-se que o custo de construção das estruturas utilizadas pelo Select é linear.

1.2.2 Utilização

Para calcular o resultado da operação Select, deve-se seguir por um dos caminhos do esquema da Figura 1.2 ajustando índices para acessar as tabelas do caminho seguido corretamente. Veja o Programa 1.13.

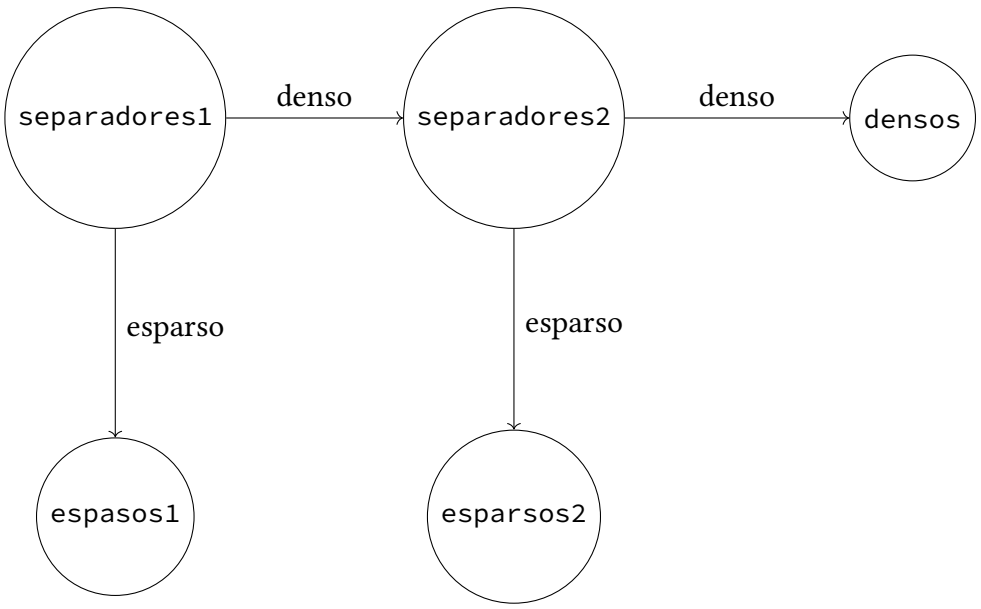

Figura 1.2: *Fluxograma de execução da operação Select.*

Programa 1.13 Cálculo do SELECT

```
1   função SELECT(k)
2       i ← k / (LS)                                          ▷ índice do setor
3       k ← k mod LS

4       se COMPRIMENTODOSETOR(i) > (LS)²                      ▷ é esparso?
5           devolva esparsos1[INÍCIODOSETOR(i) / (LS)²][k]

6       j ← k / S²                                            ▷ índice do subsetor
7       k ← k mod S²

8       se COMPRIMENTODOSUBSETOR(i, j) > S⁴                   ▷ é esparso?
9           devolva esparsos2[INÍCIODOSUBSETOR(i, j) / S⁴][k]

10      p ← INÍCIODOSUBSETOR(i, j)
11      repita
12          conteúdo ← LERDEVETORDEBITS(v, p, L)
13          se k ≤ densos[conteúdo][0]
14              devolva p + densos[conteúdo][k]
15          k ← k − densos[conteúdo][0]
16          p ← p + L
```

É simples notar que se o bit buscado não está em um subsetor denso, o SELECT é computado em tempo constante.

Por outro lado, um subsetor denso, ou seja, com comprimento menor ou igual a S^4, pode se dividir em até $\lceil \frac{S^4}{L} \rceil$ trechos de comprimento L. Como consequência, o laço das linhas 11-16 do Programa 1.13 é necessário, mas esse completa no máximo $\lceil \frac{S^4}{L} \rceil$ iterações. Observe que $\lceil \frac{S^4}{L} \rceil = O(1)$ (concretamente $\lceil \frac{S^4}{L} \rceil \leq 4$ para todo $n \geq 2$).

Desta forma conclui-se que a operação SELECT tem custo constante no modelo adotado.

1.2.3 Análise de espaço

O vetor `separadores1` necessita de $\lceil \frac{n}{LS} \rceil = \Theta\left(\frac{n}{\lg n \lg\lg n}\right)$ entradas caso todos os bits do vetor sejam 1, e cada uma dessas entradas ocupa $\Theta(\lg n)$ bits para armazenar um índice absoluto. Dessa forma `separadores1` ocupa $\Theta\left(\frac{n}{\lg\lg n}\right)$ bits.

A matriz `esparsos1` precisa comportar o número máximo de setores esparsos, isto é, ter $\left\lceil \frac{n}{(LS)^2} \right\rceil = \Theta\left(\frac{n}{(\lg n \lg\lg n)^2}\right)$ linhas e uma coluna para cada bit 1 dentro desses setores esparsos, ou seja, $LS = \Theta(\lg n \lg\lg n)$ colunas. Além disso, cada uma de suas células contém um índice absoluto para um bit 1, o que ocupa $\Theta(\lg n)$ bits. Assim conclui-se que `esparsos1` requer $\Theta\left(\frac{n}{\lg\lg n}\right)$ bits.

A matriz `separadores2` contém o número de linhas igual ao número máximo de

setores possíveis e o número de colunas igual ao número máximo de subsetores em um setor. Esses são, respectivamente, $\left\lceil \frac{n}{LS} \right\rceil = \Theta\left(\frac{n}{\lg n \lg\lg n} \right)$ e $\left\lceil \frac{LS}{S^2} \right\rceil = \left\lceil \frac{L}{S} \right\rceil = \Theta\left(\frac{\lg n}{\lg\lg n} \right)$. Como cada uma das células da matriz armazena um índice relativo a um setor, cada uma delas deve representar o comprimento do maior setor denso, ou seja, deve ocupar $\lg\left((LS)^2\right) = \Theta(\lg(\lg n \lg\lg n)) = \Theta(\lg\lg n)$. Com isso tem-se que `separadores2` ocupa $\Theta\left(\frac{n}{\lg\lg n} \right)$ bits.

Já a matriz `esparsos2` tem uma coluna para cada um dos $\left\lceil \frac{n}{S^4} \right\rceil$ subsetores esparsos possíveis e uma linha para cada um dos S^2 bits 1 compreendidos por cada um dos subsetores, ou seja, `esparsos2` tem $\Theta\left(\frac{n}{(\lg\lg n)^2} \right)$ posições. Cada uma dessas posições armazena um índice relativo ao setor, assim como as posições em `separadores2`, logo ocupa $\Theta(\lg\lg n)$ bits. Como consequência o espaço total ocupado por `esparsos2` é, em bits, $\Theta\left(\frac{n}{\lg\lg n} \right)$.

Assim como a tabela `rank_local` usada no Rank, a tabela `densos` usada no Select utiliza $\Theta(\sqrt{n} \lg n \lg\lg n)$ bits, uma vez que esta armazena $\Theta(\lg n)$ respostas para Select de $\Theta(\sqrt{n})$ possíveis trechos de comprimento L e cada uma dessas utiliza $\Theta(\lg\lg n)$ bits.

Tendo analisado todas as estruturas utilizadas para computar a operação Select, conclui-se que elas utilizam $o(n)$ bits, ou seja, a operação é calculada de forma sucinta.

Capítulo 2

Sequências Balanceadas de Parênteses

Sequências balanceadas de parênteses podem ser utilizadas para modelar várias estruturas. Como exemplo têm-se árvores enraizadas, cuja modelagem será apresentada no Capítulo 3. Nesse capítulo, um abre parêntese e um fecha parêntese, ou seja, os símbolos "(" e ")", serão chamados simplesmente de *abre* e *fecha*, respectivamente. Numa sequência balanceada de parênteses, cada abre corresponde a um fecha. Para cada abre/fecha, o seu correspondente fecha/abre será chamado de *parceiro*. Para representar uma sequência de parênteses como um vetor de bits, utilizam-se bits 1 para abres e bits 0 para fechas.

Duas operações são essenciais para navegar por sequências balanceadas de parênteses:

- $\textsc{Match}(i)$ encontra o índice do parceiro do parêntese na i-ésima posição da sequência.
- $\textsc{Enclose}(i)$ recebe o índice de um abre da sequência e encontra o índice j do abre do par de parênteses que envolve mais justamente o par $(i, \textsc{Match}(i))$, ou seja, o par que o envolve minimizando $\textsc{Match}(j) - j$.

É trivial construir uma estrutura não sucinta que dá suporte a essas operações. Basta iterar sobre a sequência mantendo uma pilha com as posições dos abres ainda não fechados e catalogar Match de todos os parênteses e o Enclose de todos os abres. Veja o Programa 2.1.

É simples notar que o custo de construção dessa estrutura é linear e as tabelas `matchTrivial` e `encloseTrivial` ocupam $O(n \lg n)$ bits.

Para dar suporte a Match e Enclose sucintamente, Geary, Rahman, Raman e Raman [Gea+06] mantêm uma subestrutura recursiva de subsequências balanceadas da sequência original, uma coleção de tabelas específicas para cada uma das operações, além de tabelas triviais, como as construídas pelo Programa 2.1, para a menor das subsequências mantidas, se essa tiver tamanho maior que $\left\lceil \frac{\lg n}{2} \right\rceil$. A subestrutura recursiva utiliza o conceito introduzido na próxima seção.

Programa 2.1 Construção de estrutura não sucinta para MATCH e ENCLOSE

```
1   função ConstróiTrivial(v[1..n])
2       pilha ← NovaPilha()
3       para i ← 1 até n
4           se v[i]                                          ▷ é um abre
5               se Tamanho(pilha) > 0
6                   encloseTrivial[i] ← Topo(pilha)
7               Empilha(pilha, i)
8           senão
9               parceiro ← Desempilha(pilha)
10              matchTrivial[i] ← parceiro
11              matchTrivial[parceiro] ← i
12      devolva (matchTrivial, encloseTrivial)
```

2.1 Famílias de pioneiros

O conceito de família de pioneiros depende de um parâmetro L. Considere uma sequência balanceada de parênteses de comprimento N. Geary, Rahman, Raman e Raman [Gea+06] dividem essa sequência em blocos de tamanho L. Para fins de concisão, defina β como o número de blocos em que a sequência foi dividida, ou seja, $\beta = \left\lceil \frac{N}{L} \right\rceil$. Defina também duas funções: $b(x)$ como o bloco a que o parêntese x pertence e $m(x)$ como o parceiro de x. Tendo essas definições, utiliza-se de três classificações para um parêntese, assim como apresentadas por Jacobson [Jac88]:

- *Próximo*: Um parêntese x é dito próximo se $b(x) = b(m(x))$.
- *Distante*: Inversamente, um parêntese x é chamado de distante se $b(x) \neq b(m(x))$.
- *Pioneiro*: Um abre distante x é classificado como pioneiro quando é o primeiro abre distante da sequência ou quando $b(m(x)) \neq b(m(y))$, onde y é o abre distante que o precede. Para um fecha distante, a definição é similar, porém tem-se como referência o fecha distante que o sucede. Veja a Figura 2.1.

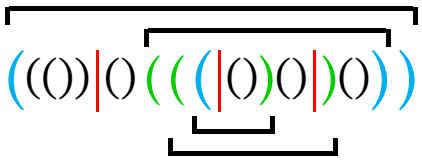

Figura 2.1: *Classificação de parênteses: os parênteses azuis (maiores) e verdes (de tamanho médio) são distantes; os azuis são pioneiros. Os traços vermelhos representam a divisão em blocos.*

Note que é possível que o parceiro de um parêntese pioneiro não seja pioneiro. Veja o segundo abre pioneiro da Figura 2.1.

Partindo dessas classificações, Geary, Rahman, Raman e Raman [Gea+06] definem uma *família de pioneiros* como o conjunto de todos os (parênteses) pioneiros unido do conjunto de seus parceiros. É simples notar que a sequência formada somente pelos parênteses contidos em uma família de pioneiros é balanceada.

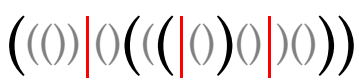

Figura 2.2: *Os parênteses em destaque formam a família de pioneiros da sequência.*

É possível delimitar o tamanho da família de pioneiros por meio da criação de um grafo com vértices $1, \ldots, \beta$ e, para cada par de parênteses $\{x, m(x)\}$ da família, uma aresta entre $b(x)$ e $b(m(x))$.

Um grafo é classificado como *exoplanar* se pode ser desenhado no plano de forma que nenhuma de suas arestas se cruzem e todos os vértices façam parte da fronteira da face exterior [Die17].

Figura 2.3: *Grafo da família de pioneiros para a sequência da Figura 2.2.*

Note que o grafo criado da família de pioneiros é exoplanar, uma vez que pode ser desenhado com todos os vértices em uma linha e todas as arestas acima desta linha, sem cruzamentos. Como o grafo é exoplanar e não contém arestas paralelas, pois a família é de pioneiros, esse contém no máximo $2\beta - 3$ arestas. Uma vez que, no grafo, existe uma aresta para cada par de parênteses da família, conclui-se que essa tem tamanho, no máximo, $4\beta - 6$.

Há uma propriedade sobre parênteses pioneiros e suas famílias que permite a identificação simples dos parênteses que compõem a família de pioneiros:

Para todo fecha pioneiro x, seu abre $m(x)$ é pioneiro ou é o primeiro abre distante de seu bloco. (Veja a Figura 2.1.) De fato, suponha que exista um fecha pioneiro x tal que $m(x)$ não é pioneiro e não é o primeiro abre distante de seu bloco. Seja y o abre distante que precede $m(x)$. Como $m(x)$ não é pioneiro, então $b(x) = b(m(y))$. Desta forma, o par $(y, m(y))$ envolve o par $(m(x), x)$, o que faz com que $m(y)$ seja o fecha distante que sucede x. Por outro lado, $m(x)$ não é o primeiro distante de seu bloco, logo $b(m(x)) = b(y)$. Contudo, isso implica que x não é pioneiro, uma contradição.

Com essa propriedade, é possível identificar os índices de todos os parênteses que compõem a família de pioneiros a partir de uma lista ordenada de pares de índices de abres distantes e seu parceiros, como feito no Programa 2.2.

Essa função tem custo $\mathrm{O}(N) + \mathrm{O}(\beta \lg \beta) = \mathrm{O}(N) + \mathrm{O}\left(\frac{N}{L} \lg \frac{N}{L}\right)$, onde o segundo termo é o custo da ordenação dos índices dos parênteses que compõem a família. Isso é $\mathrm{O}(N)$ desde que $L = \Omega(\lg N)$.

2.2 Subestrutura recursiva

Para dar suporte a Match e Enclose sucintamente, Geary, Rahman, Raman e Raman [Gea+06] propuseram o uso de uma estrutura de níveis construída recursivamente. O

Programa 2.2 Identificação de uma família de pioneiros

Recebe L.

```
1   função B(i)                                                  ▷ bloco de i
2       devolva ⌈i/L⌉

3   função FamíliaDePioneiros(v[1..N])
4       p ← NovaPilha()
5       distantes ← NovaLista()
6       para i ← 1 até N
7           se v[i]                                              ▷ é um abre
8               Empilha(p, i)
9           senão
10              mi ← Desempilha(p)
11              se B(i) ≠ B(mi)
12                  Insere(distantes, (mi, i))
13      família ← NovaLista()
14      para i ← 1 até Tamanho(distantes)
15          (x, mx) ← distantes[i]
16          se i ≠ 1
17              (y, my) ← distantes[i − 1]
18              se B(x) = B(y) e B(mx) = B(my)
19                  continue
20          Insere(família, x)
21          Insere(família, mx)
22      Ordene(família)
23      devolva família
```

primeiro nível (nível 0) da estrutura é a sequência de parênteses original. Cada um dos níveis seguintes da estrutura é a sequência de parênteses correspondente à família de pioneiros da sequência do nível anterior, acrescida de uma estrutura de dados descrita a seguir. Para construção das famílias de pioneiros utiliza-se o parâmetro $L = \left\lceil \frac{\lg n}{2} \right\rceil$.

Para guardar informação sobre a família de pioneiros que deu origem à sequência de parênteses de cada nível, é utilizada uma estrutura conhecida como Dicionário Completamente Indexável (DCI). Essa estrutura, construída a partir de um inteiro M e um conjunto $S \subseteq [M]$, permite quatro operações:

- Rank e Select, de forma semelhante às vistas no Capítulo 1, devolvem, respectivamente, quantos elementos de S precedem um elemento de $[M]$, e o i-ésimo menor elemento de S, onde $i \in [|S|]$.
- Pred e Suc recebem um $x \in [M]$ e devolvem, respectivamente, $\max\{y \in S : y \leq x\}$ e $\min\{y \in S : y \geq x\}$.

Raman, Raman e Satti [RRS07] apresentam uma implementação de um DCI que utiliza no máximo

$$\left\lceil \lg \binom{M}{|S|} \right\rceil + O\left(\frac{M \lg\lg M}{\lg M} \right) \text{ bits.} \tag{2.1}$$

Com isso, exceto pelo primeiro nível, cada nível da subestrutura recursiva é composto por:

- Uma sequência contendo somente os parênteses da família de pioneiros do nível anterior.
- Um DCI para a família de pioneiros do nível anterior.

Usa-se N para denotar o comprimento da sequência de parênteses em um determinado nível da recursão. É evidente que caso $N \leq L$ a recursão termina, uma vez que a sequência tem apenas um bloco, ou seja, todos os parênteses são próximos e a família de pioneiros é vazia. Nas Seções 2.3 e 2.4 serão apresentadas implementações recursivas de Match e Enclose que se baseiam nos níveis dessa estrutura. Para que essas operações possam ter custo O(1) será necessário que exista um número constante de níveis na estrutura recursiva. Isso decorre do fato de que uma dessas operações poderá acessar todos os k níveis existentes na estrutura e nesse caso seu custo será $\Omega(k)$. Por isso será escolhido um valor constante para k e no caso em que a estrutura termina ao atingir o número máximo k de níveis, mas ainda contendo mais que um bloco, é adicionada à estrutura uma implementação trivial, assim como a descrita no Programa 2.1, para a última sequência de parênteses.

O Programa 2.3 apresenta como construir os vetores `DCI` e `sequênciaNível`, que mantêm, respectivamente, os DCIs e as subsequências de cada um dos k níveis da subestrutura recursiva. Para isso, utiliza-se das funções **ConstróiDCI**, que constrói um DCI ao receber um inteiro M e um vetor de valores que representa um conjunto $S \subseteq [M]$, e **Subsequência**, que constrói um novo vetor de bits a partir de um vetor de bits e uma lista de índices de bits a serem copiados, além de **ConstróiTrivial**.

Programa 2.3 Construção da subestrutura recursiva com k níveis

```
1   função N(K)
2       devolva Tamanho(sequênciaNível[K])

3   função ConstróiSubestruturaRecursiva(sequênciaOriginal)
4       K ← 0
5       sequênciaNível[K] ← sequênciaOriginal
6       enquanto K < k e N(K) > L
7           família ← FamíliaDePioneiros(sequênciaNível[K])
8           DCI[K] ← ConstróiDCI(N(K), família)
9           sequênciaNível[K + 1] ← Subsequência(sequênciaNível[K], família)
10          K ← K + 1
11      se K = k
12          matchTrivial, encloseTrivial ← ConstróiTrivial(sequêncialNível[K])
```

O custo de construção da estrutura é dominado pela construção do primeiro nível. Desta forma essa tem custo esperado linear, uma vez que a construção do DCI descrita por Raman, Raman e Satti [RRS07] depende de um algoritmo aleatório para encontrar uma função de hash perfeita.

Analisando o espaço necessário para armazenar cada um dos níveis da estrutura, tem-se

que cada um desses é composto pela subsequência formada pelos parênteses da família de pioneiros, cujo tamanho é no máximo $4\frac{N}{L} - 6$, e um DCI com parâmetros: $M = N$ e S como a família de pioneiros, cujo tamanho é $|S| \leq 4\left\lceil \frac{N}{L} \right\rceil - 6$. A partir da delimitação $\binom{a}{b} < \left(\frac{ea}{b}\right)^b$, decorre da Equação 2.1 e de $L = \Theta(\lg n)$ que o espaço necessário para cada nível é $\mathrm{O}\left(\frac{N \lg\lg N}{\lg N}\right)$ bits.

Como no último nível da estrutura é mantida uma implementação trivial, como a construída pelo Programa 2.1, tem-se que o espaço necessário para a subestrutura recursiva é $S(n, 0)$, onde n é o tamanho da sequência original e $S(N, K)$ se dá pela recorrência:

$$S(N,K) = \begin{cases} S(4\left\lceil \frac{N}{L} \right\rceil - 6, K+1) + \mathrm{O}\left(\frac{N \lg\lg N}{\lg N}\right) & \text{para } K < k \\ \Theta(N \lg N) & \text{se } K = k. \end{cases}$$

Expandindo esta recorrência, desde que $k > 1$, obtém-se que $S(n, 0) = \mathrm{O}\left(\frac{n \lg\lg n}{\lg n}\right) + \mathrm{O}\left(\frac{n}{\lg^{k-1} n}\right)$. Assim, para que esta subestrutura seja sucinta, ou seja, $\mathrm{O}\left(\frac{n \lg\lg n}{\lg n}\right) + \mathrm{O}\left(\frac{n}{\lg^{k-1} n}\right) = \mathrm{o}(n)$ é necessário que $k \geq 2$.

Com todos os níveis da subestrutura recursiva construídos, basta descrever como as tabelas específicas para cada uma das operações são construídas e utilizadas, junto da subestrutura recursiva, para computar Match e Enclose.

2.3 Match

Durante a operação Match para um parêntese qualquer x, busca-se por seu parceiro $m(x)$. Para o cálculo sucinto dessa operação são criadas cinco tabelas de consulta, que mantêm somente informação local ao bloco para qualquer bloco: `parceiro`, `excessoEsquerdo`, `excessoDireito`, `primeiroComExcesso` e `últimoComExcesso`.

A tabela `parceiro` mantém, para cada parêntese, a posição de seu parceiro se esse estiver no bloco, ou um valor sentinela, caso contrário.

A tabela `excessoEsquerdo` contém, para cada posição do bloco, o excesso esquerdo relativo ao início do bloco, isto é, a quantidade de abres menos a quantidade de fechas do início do bloco até a posição. De forma semelhante, `excessoDireito` mantém, para cada posição do bloco, o excesso direito, quantidade de fechas menos a quantidade de abres, relativo ao fim do bloco.

Já a tabela `primeiroComExcesso` armazena, para cada excesso direito relativo possível e no bloco, a posição do primeiro fecha do bloco com excesso direito e. Note que $e \in [-L; L]$, assim, por simplicidade de notação, essa tabela será indexada por esse intervalo. Como a nomenclatura sugere, `últimoComExcesso` é similar a `primeiroComExcesso`, mantendo a posição do último abre do bloco com cada excesso esquerdo possível no bloco.

Por evitar repetições, omitiu-se a implementação de Match para um fecha. Consequentemente, também foram omitidas a construção, utilização e análise de `últimoComExcesso`, tabela utilizada somente nessa ocasião.

Programa 2.4 Cálculo da operação MATCH

Caso em que v`[i]` é um abre.

```
1   função Match(i)
2       devolva MatchRec(i, 0)

3   função MatchRec(i, K)
4       se K = k e N(K) > L
5           devolva matchTrivial[i]
6       senão
7           conteúdo ← LerBloco(K, B(i))
8           iRelativo ← i mod L
9           se parceiro[conteúdo][iRelativo] ≠ −1
10              devolva (B(i) − 1)L + parceiro[conteúdo][iRelativo]
11          j ← Pred(DCI[K], i)
12          jsubnível ← Rank(DCI[K], j)
13          mjsubnível ← MatchRec(jsubnível, K + 1)
14          mj ← Select(DCI[K], mjsubnível)
15          mconteúdo ← LerBloco(K, B(mj))
16          dExc ← excessoEsquerdo[conteúdo][j] − excessoEsquerdo[conteúdo][i]
17          exc ← excessoDireito[conteúdo][mj] + dExc
18          devolva (B(mj) − 1)L + primeiroComExcesso[mconteúdo][exc]
```

Utilizando essas tabelas, o cálculo recursivo da operação MATCH, assim como no Programa 2.4, começa verificando se o parêntese x da posição i é próximo por meio da tabela `parceiro`. Se x é próximo, o algoritmo termina uma vez que a posição de $m(x)$ foi encontrada (linhas 7-10).

Caso contrário, ou seja, se x é distante, decorre da definição da família de pioneiros que existe nessa um par de parênteses $(y, m(y))$, tal que $b(x) = b(y)$ e $b(m(x)) = b(m(y))$, que pode auxiliar na busca por $m(x)$. Para encontrar y, utiliza-se o DCI, uma vez que y é o parêntese da família de pioneiros que precede ou é x. Com a posição de y, faz-se uma chamada recursiva a MATCH para y e a família de pioneiros para encontrar a posição de $m(y)$ (linhas 11-14).

Como a sequência é balanceada, tem-se que a variação de excesso esquerdo entre y e x é igual à variação de excesso direito entre $m(x)$ e $m(y)$. Com isso, é possível utilizar dois acessos a `excessoEsquerdo` para calcular a variação de excesso esquerdo entre y e x e um acesso a `excessoDireito` para calcular qual é o excesso direito de $m(x)$ no bloco $b(m(y))$ (linhas 15-17).

Note que $m(x)$ é o primeiro fecha do bloco com tal excesso, uma vez que todos os fechas subsequentes com o mesmo excesso direito devem ser próximos. Veja a Figura 2.4. Com isso, basta fazer um acesso a `primeiroComExcesso` para encontrar a posição de $m(x)$, concluindo a execução do algoritmo (linha 18).

Para que a implementação de MATCH fique mais simples, pode-se definir uma subrotina chamada **LerBloco**. Essa subrotina recebe o identificador de nível da estrutura recursiva K e um índice de bloco `b`, devolvendo o conteúdo do `b`-ésimo bloco da sequência de

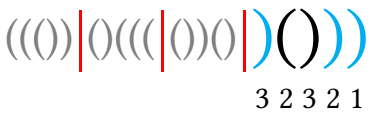

Figura 2.4: *Excesso direito em um bloco: parênteses em azul são distantes.*

parênteses do K-ésimo nível da estrutura recursiva, possivelmente acrescido de zeros à direita para completar o comprimento L. Como cada bloco tem comprimento $L = \mathrm{O}(\lg n)$, a rotina **LerBloco** faz somente uma chamada à primitiva **LerDeVetorDeBits**, ou seja, tem custo constante no modelo.

Com isso, pode-se computar o Match de um abre com custo constante no modelo, conforme o Programa 2.4.

As tabelas de consulta utilizadas para o Match são de simples construção. As tabelas `parceiro` e `excessoEsquerdo` podem ser construídas ao iterar sobre o conteúdo do bloco mantendo uma pilha para encontrar os pares de parênteses e um inteiro `exc` que corresponde ao excesso relativo. A construção dessas duas tabelas tem custo $\mathrm{O}(\sqrt{n}\lg n)$ no modelo, conforme o Programa 2.5.

Programa 2.5 Construção de uma linha das tabelas `parceiro` e `excessoEsquerdo`

```
1   função ConstróiParceiroEExcessoEsquerdo(conteúdo)
2     p ← NovaPilha()
3     exc ← 0
4     para i ← 1 até L
5       parceiro[conteúdo][i] ← -1              ▷ valor sentinela para distantes
6     para i ← 1 até L
7       se conteúdo[i]
8         Empilha(p, i)
9         exc ← exc + 1
10      senão
11        se exc > 0
12          mi ← Desempilha(p)
13          parceiro[conteúdo][mi] ← i
14          parceiro[conteúdo][i] ← mi
15        exc ← exc - 1
16      excessoEsquerdo[conteúdo][i] ← exc
```

A construção de `excessoDireito` e `primeiroComExcesso` também pode ser feita simultaneamente por meio de uma iteração sobre o conteúdo do bloco na ordem inversa. Assim, o custo de construção destas tabelas também é $\mathrm{O}(\sqrt{n}\lg n)$. Veja o Programa 2.6.

Quanto ao espaço, é simples notar que as tabelas consomem espaço $\mathrm{O}(\sqrt{n}\lg n \lg\lg n)$ uma vez que cada uma de suas $\sqrt{n}$ linhas tem, no máximo, $2L = \mathrm{O}(\lg n)$ posições e cada uma dessas armazena números no intervalo $[-L; L]$, ou seja, consomem espaço $\mathrm{O}(\lg L) = \mathrm{O}(\lg\lg n)$.

Com isso, conclui-se que as tabelas utilizadas para calcular a operação Match consomem espaço $\mathrm{O}(\sqrt{n}\lg n \lg\lg n) = \mathrm{o}(n)$.

Programa 2.6 Construção das tabelas `excessoDireito` e `primeiroComExc`

```
1  função ConstróiExcessoDireitoEPrimeiroComExc(conteúdo)
2      exc ← 0
3      para i ← L descendo até 1
4          se conteúdo[i]
5              exc ← exc − 1
6          senão
7              exc ← exc + 1
8          excessoDireito[conteúdo][i] ← exc
9          primeiroComExcesso[conteúdo][exc] ← i
```

2.4 Enclose

O objetivo da operação Enclose para um abre qualquer x é encontrar o abre y cujo par $(y, m(y))$ envolve $(x, m(x))$ mais justamente. Por simplicidade, escrevemos que y é o abre que envolve x mais justamente e $m(y)$ é o fecha que envolve x mais justamente.

Para o cálculo sucinto da operação Enclose são criadas três tabelas de consulta, ou seja, tabelas que contém somente informação local ao bloco para qualquer bloco possível: `envolve`, `distantePredecessor` e `últimoDistante`.

A tabela de consulta `envolve` mantém, para cada abre x do bloco, a posição no bloco em que o abre ou o fecha que envolve x mais justamente reside; se nenhum desses estiverem no bloco, `envolve` mantém um valor sentinela.

A tabela de consulta `distantePredecessor` mantém, para cada abre do bloco, a posição do último abre distante que o precede. Por fim, `últimoDistante` mantém um único valor por bloco: a posição de seu último abre distante.

A partir dessas tabelas é possível calcular Enclose recursivamente com o seguinte método. Acompanhe com o Programa 2.7.

Seja x o abre da posição i. Para encontrar o abre y que envolve x mais justamente, inicialmente verifica-se se y ou $m(y)$ estão no bloco $b(x)$ por meio de um acesso à tabela `envolve`. Se y ou $m(y)$ estão no bloco $b(x)$, então a posição de y pode ser encontrada por meio de, no máximo, uma chamada à Match (linhas 9-14). Veja a Figura 2.5.

Figura 2.5: *Os parênteses em preto (de tamanho médio) representam x e $m(x)$, já os em azul (maiores) representam y e $m(y)$. Na sequência da esquerda, são destacados dois pares $(x, m(x))$. No primeiro deles y e $m(y)$ estão no mesmo bloco que x, já no segundo x não compartilha bloco com y nem com $m(y)$. A sequência da direita apresenta dois exemplos onde x compartilha bloco com somente um dos parênteses que o envolvem mais justamente.*

Caso nem y, nem $m(y)$ estejam no bloco $b(x)$, então pode-se afirmar que $b(y) < b(x) < b(m(y))$, ou seja, y é distante. Como y é distante, tem-se que o par $(z, m(z))$ da família de pioneiros que envolve x mais justamente compartilha bloco com y e $m(y)$, ou seja,

Programa 2.7 Cálculo da operação ENCLOSE

```
1   função Enclose(i)
2       devolva EncloseRec(i, 0)

3   função EncloseRec(i, K)
4       se K = k e N(K) > L
5           devolva encloseTrivial[i]
6       senão
7           conteúdo ← LerDeBloco(K, B(i))
8           iRelativo ← i mod L
9           se envolve[conteúdo][iRelativo] ≠ −1          ▷ y ou m(y) está no bloco
10              y ← (B(i) − 1)L + envolve[conteúdo][iRelativo]
11              se sequênciaNível[K][y]
12                  devolva y
13              senão
14                  devolva MatchRec(y, K)
15          p ← Suc(DCI[K], i)
16          se sequênciaNível[K][p]
17              psubnível ← Rank(DCI[K], p)
18              zsubnível ← EncloseRec(psubnível, K + 1)
19              z ← Select(DCI[K], zsubnível)
20          senão
21              z ← MatchRec(p, K)
22          w ← Suc(DCI[K], z + 1)
23          conteúdo ← LerBloco(K, B(z))
24          inícioDoBloco ← (B(z) − 1)L
25          se B(z) = B(w)
26              wRelativo ← w mod L
27              devolva inícioDoBloco + distantePredecessor[conteúdo][wRelativo]
28          senão
29              devolva inícioDoBloco + últimoDistante[conteúdo]
```

$b(z) = b(y)$ e $b(m(y)) = b(m(z))$. Note que se y ou $m(y)$ forem pioneiros, $y = z$, contudo não há um teste simples para verificar se esse é o caso. De qualquer forma, z nos auxiliará a encontrar y.

Com o intuito de encontrar z, utiliza-se o DCI para encontrar o parêntese p da família de pioneiros que sucede ou é x. Se p é um abre, então o abre que envolve p mais justamente na família de pioneiros também é z, logo esse pode ser encontrado acionando ENCLOSE recursivamente para p e a família de pioneiros (linhas 15-19). Veja o exemplo esquerdo da Figura 2.6. Caso contrário, ou seja, se p é um fecha, então $p = m(z)$ (linha 21). Veja o exemplo direito da Figura 2.6.

Figura 2.6: *Exemplos de busca por z. Os parênteses em azul representam a família de pioneiros.*

Note que, uma vez que $b(z) = b(y) < b(x) < b(m(y)) = b(m(z))$, ou y é z, como ressaltado anteriormente, ou y é o último abre distante do bloco cujo predecessor da família de pioneiros é z. De fato, se y não for esse, então existe outro par de parênteses que envolve x mais justamente que y.

Com isso, para encontrar y e concluir a execução do algoritmo, utiliza-se novamente o DCI, desta vez para encontrar o sucessor w de z na família de pioneiros. Se $b(w) = b(z)$, então y é o último abre distante que precede w, e pode ser encontrado com um acesso à tabela `distantePredecessor`. Veja o exemplo da direita da Figura 2.7. Se $b(w) \neq b(z)$, então y é o último abre distante do bloco $b(z)$, logo sua posição está armazenada na tabela `últimoDistante` (linhas 22-29). Veja o exemplo da esquerda da Figura 2.7.

Figura 2.7: *Exemplos de busca por y partindo de z. Ocasionalmente, no exemplo esquerdo $y = z$.*

Com esse método, a operação ENCLOSE pode ser calculada em tempo constante, conforme o Programa 2.7.

Resta apresentar como as três tabelas de consulta utilizadas podem ser construídas. Cada uma dessas tabelas tem $2^L = O(\sqrt{n})$ linhas. Para popular uma linha de `envolve` é necessário fazer duas passadas na subsequência de parênteses à qual a linha se refere, uma procurando por abres e outra por fechas que envolvem mais justamente cada abre do trecho. Desta forma, custo linear sobre o tamanho do bloco, ou seja, $O(L) = O(\lg n)$ é necessário para popular cada linha de `envolve`, e a construção completa dessas tem custo $O(\sqrt{n} \lg n)$. Veja o Programa 2.8.

Para preencher as tabelas de consulta `distantePredecessor` e `últimoDistante`, basta iterar sobre o conteúdo do bloco na ordem inversa e ao encontrar um abre distante preencher as posições necessárias. Como apresenta o Programa 2.9, essas construções têm custo $O(\sqrt{n} \lg n)$.

Ao analisar o espaço necessário para armazenar `envolve` e `distantePredecessor`, vê-se que ambas utilizam espaço $O(\sqrt{n} \lg n \lg \lg n)$, já que armazenam uma posição relativa ao bloco, ou seja, de tamanho $O(\lg \lg n)$ para cada uma das $L = O(\lg n)$ posições dos $2^L = O(\sqrt{n})$ blocos possíveis. Por fim, `últimoDistante` requer somente $O(\sqrt{n} \lg \lg n)$ bits, já que mantém somente um valor por bloco.

Desta forma, vê-se que a estrutura apresentada por Geary, Rahman, Raman e Raman [Gea+06] utiliza espaço adicional $o(n)$, ou seja, é sucinta e disponibiliza implementações de custo constante para as operações MATCH e ENCLOSE.

Programa 2.8 Construção de uma linha da tabela `envolve`

Recebe L e um vetor de bits `conteúdo`$[1..L]$

```
1   pilha ← NovaPilha()
2   para i ← 1 até L
3      se conteúdo[i]
4         se Tamanho(pilha) ≠ 0
5            envolve[conteúdo][i] ← Topo(pilha)
6         senão
7            envolve[conteúdo][i] ← −1
8         Empilha(pilha, i)
9      senão
10        se Tamanho(pilha) ≠ 0
11           Desempilha(pilha)

12  pilha ← NovaPilha()
13  para i ← L descendo até 1
14     se conteúdo[i]
15        se Tamanho(pilha) ≠ 0
16           Desempilha(pilha)
17           se Tamanho(pilha) ≠ 0
18              envolve[conteúdo][i] ← Topo(pilha)
19     senão
20        Empilha(pilha, i)
```

Programa 2.9 Construção das tabelas `distantePredecessor` e `últimoDistante`

Recebe L e um vetor de bits `conteúdo`$[1..L]$

```
1   para i ← 1 até L
2      distantePredecessor[conteúdo][i] ← −1
3   últimoDistante[conteúdo] ← −1
4   c ← 0
5   para i ← L descendo até 1
6      se conteúdo[i]
7         se c = 0
8            se últimoDistante[conteúdo] = −1
9               últimoDistante[conteúdo] ← i
10           j ← i + 1
11           enquanto j ≤ L e distantePredecessor[conteúdo][j] = −1
12              distantePredecessor[conteúdo][j] ← i
13              j ← j + 1
14        c ← c − 1
15     senão
16        c ← c + 1
```

Capítulo 3

Árvores Binárias Sucintas

Uma das estruturas de dados mais comuns da computação são as árvores. Essas são utilizadas, por exemplo, em representações eficientes de conjuntos e sequências, além de servirem de base de várias outras estruturas de dados mais sofisticadas. As estruturas apresentadas nos Capítulos 1 e 2 servem como base para duas representações sucintas distintas de árvores binárias estáticas.

Essas representações suportam eficientemente as seguintes operações sobre a árvore representada:

- **TemFilhoEsquerdo** e **TemFilhoDireito** que recebem o identificador de um nó e devolvem um booleano referente à existência dos filhos correspondentes desse nó.
- **FilhoEsquerdo** e **FilhoDireito** que recebem o identificador de um nó e devolvem o identificador do filho requisitado.
- **Pai** que recebe o identificador de um nó e devolve o identificador do nó pai desse nó.

As duas representações diferem quanto à simplicidade e à capacidade de prover uma implementação para a operação **Tamanho** com custo constante. Essa operação devolve o número de nós da subárvore de um dado nó. Enquanto a representação de árvore binária de Clark [Cla97] é muito mais simples e não provê suporte eficiente para a operação **Tamanho**, a representação de Munro e Raman [MR97] permite essa operação, porém é bastante complexa, uma vez que, como veremos, representa indiretamente a árvore binária como uma árvore enraizada ordenada.

Para saber se as representações são sucintas, é imprescindível conhecer delimitações sobre o número de bits necessários para representar árvores binárias de n nós. Há uma expressão que representa a quantidade de árvores binárias com n nós. Essa expressão é o n-ésimo *número de Catalan*: $C_n = \frac{1}{2n+1}\binom{2n}{n}$ [Sta99].

Utilizando a aproximação de Stirling para o logaritmo de fatoriais, decorre que $C_n \approx 4^n$. Dessa forma, representações de árvores requerem $\lg C_n \approx 2n$ bits e, com isso, representações de árvores binárias que utilizam $2n + o(n)$ bits são sucintas.

3.1 Árvores binárias baseadas em Rank e Select

Clark [Cla97] apresenta uma representação obtida ao percorrer os nós de uma árvore binária ordenados por seu nível. A representação consiste em uma sequência de bits. Para a construção dessa sequência, inicialmente, todos os nós da árvore são marcados com 1. Em seguida, são adicionadas folhas marcadas com 0 de modo que todos os nós originais da árvore sejam internos e tenham dois filhos. Na última etapa da construção da sequência, faz-se um percurso visitando os nós como no algoritmo de busca em largura, sempre visitando primeiro a subárvore esquerda de cada nó, depois a direita, inserindo os bits marcadores assim que são encontrados. Com isso, a sequência de bits formada tem comprimento $2n + 1$ onde n é número de nós da árvore original. Veja a Figura 3.1.

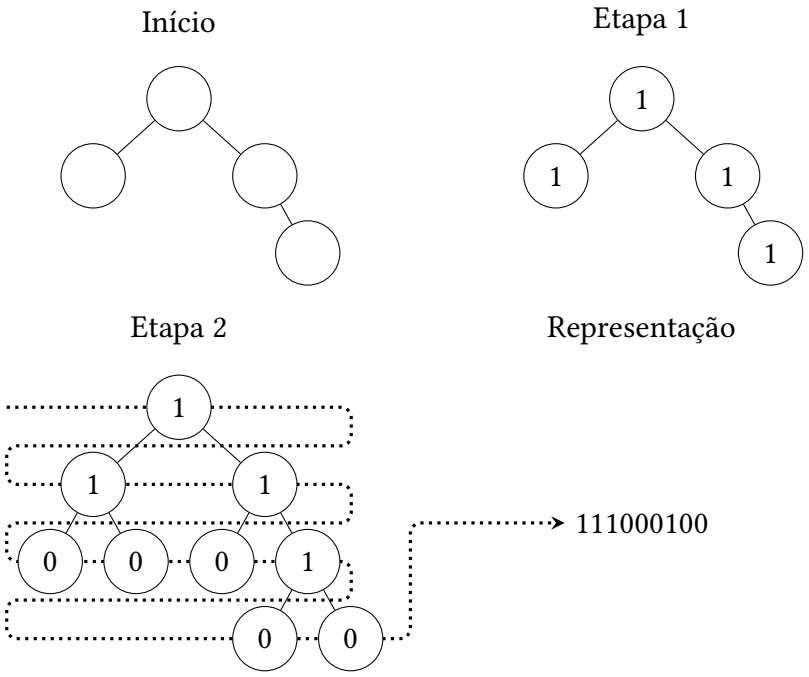

Figura 3.1: *Passo a passo da construção da sequência de bits que corresponde a uma dada árvore.*

Essa construção pode ser feita de forma direta e com custo linear, assim como no Programa 3.1, que considera que cada nó da árvore tem campos `esq`, `dir` e `tam`, que indicam o filho esquerdo, o direito e o número de nós de subárvore do nó.

Se o bit 1 que representa um nó da árvore é o i-ésimo bit 1 da sequência gerada, nos referimos a este como o i-ésimo nó da árvore original e a i como seu identificador. A partir disso, tem-se que os bits da sequência que representam os filhos do i-ésimo nó estão nas posições $2i$ e $2i+1$ da sequência gerada, uma vez que o primeiro bit da sequência representa a raiz e para cada um dos $i - 1$ bits 1 que precedem o i-ésimo bit 1 existem dois bits que precedem os bits que representam os filhos do i-ésimo nó. Ademais, o identificador de um nó cujo bit 1 está na posição j pode ser encontrado se soubermos quantos bits 1 existem na sequência até a posição j, ou seja, seu identificador é Rank(j). Com isso, tem-se um algoritmo para descer na árvore utilizando somente a operação Rank, assim como nas primeiras quatro rotinas do Programa 3.2.

Programa 3.1 Construção da representação de Clark

```
1   função RepresentaçãoOrdemDeNível(raiz)
2       v ← vetor de 2·raiz.tam + 1 bits
3       i ← 1
4       fila ← NovaFila()                          ▷ fila de nós
5       Insere(fila, raiz)
6       enquanto Tamanho(fila) ≠ 0
7           nó ← Remove(fila)
8           se nó ≠ null
9               v[i] = 1
10              Insere(fila, nó.esq)
11              Insere(fila, nó.dir)
12          senão
13              v[i] = 0
14          i ← i + 1
15      devolva v
```

Para subir na árvore, ou seja, implementar a operação **Pai**, basta utilizar operações inversas às utilizadas na descida, ou seja, Select e uma divisão, como na última rotina do Programa 3.2.

Programa 3.2 Navegação na representação de Clark

```
1   função TemFilhoEsquerdo(id)
2       devolva v[2·id]

3   função TemFilhoDireito(id)
4       devolva v[2·id + 1]

5   função FilhoEsquerdo(id)
6       devolva Rank(2·id)

7   função FilhoDireito(id)
8       devolva Rank(2·id + 1)

9   função Pai(id)
10      devolva Select(id) / 2
```

Note que as operações são descritas com base nas operações Rank e Select, apresentadas no Capítulo 1, logo podem ser calculadas com custo constante utilizando espaço adicional $o(n)$. Com isso, conclui-se que essa representação da árvore binária é sucinta, uma vez que utiliza $2n + o(n)$ bits.

3.2 Árvores binárias baseadas em sequências balanceadas de parênteses

A representação de Munro e Raman [MR97] se baseia numa bijeção entre árvores binárias e árvores enraizadas ordenadas. Seja T uma árvore binária. Nessa bijeção, a árvore enraizada que corresponde a T tem como raiz um nó artificial cujo primeiro filho é o nó que representa a raiz de T. A partir da raiz de T, o filho direito de cada nó x de T é transformado no irmão seguinte do nó que corresponde a x na árvore enraizada ordenada. Já o filho esquerdo de x é representado como o primeiro filho do nó que corresponde a x na árvore enraizada. Veja um exemplo dessa bijeção na Figura 3.2

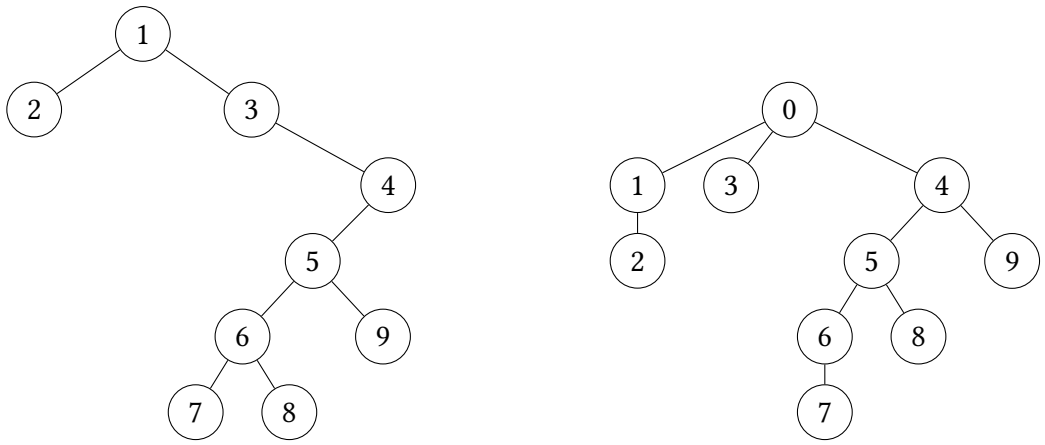

Figura 3.2: *Uma árvore binária e sua correspondente árvore enraizada.*

A partir dessa árvore enraizada, é formada a sequência balanceada de parênteses que representa a árvore original. A construção dessa sequência se dá por meio de um percurso da árvore enraizada ordenada. Começa-se com a sequência vazia e esse percurso adiciona à sequência um abre parêntese no momento em um nó é visitado pela primeira vez; em seguida, visita as subárvores dos seus filhos recursivamente e, após todas essas visitas, é adicionado à sequência um fecha parêntese. Veja a Figura 3.3.

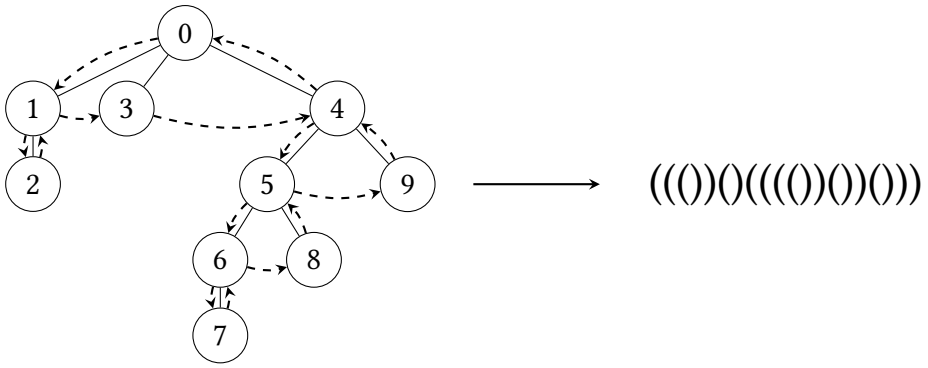

Figura 3.3: *Percurso na árvore enraizada formando a sequência utilizada nos exemplos do Capítulo 2.*

Note que o percurso descrito se assemelha a um passeio Euleriano, diferenciando-se somente por não revisitar o nó pai entre visitas de nós irmãos. Além disso, note que pode-se

construir a mesma sequência formada por esse percurso por meio de um percurso na árvore original, assim como feito no Programa 3.3.

Programa 3.3 Construção da representação de Munro e Raman

```
1  função RepresentaçãoSequênciaBalanceadaDeParênteses(raiz)
2      k ← 2 · raiz.tam + 2                    ▷ comprimento da sequência de bits
3      v ← vetor de k bits
4      v[1] ← 1                                ▷ abre artificial
5      v[k] ← 0                                ▷ fecha artificial
6      i ← 2
7      pilha ← NovaPilha()                     ▷ pilha de pares (bit, nó)
8      Empilha(pilha, (1, raiz))
9      enquanto Tamanho(pilha) ≠ 0
10         (bit, nó) ← Desempilha(pilha)
11         v[i] ← bit
12         i ← i + 1
13         se bit = 1
14             se nó.dir ≠ null
15                 Empilha(pilha, (1, nó.dir))
16             Empilha(pilha, (0, nó))
17             se nó.esq ≠ null
18                 Empilha(pilha, (1, nó.esq))
```

Por fim, é necessário descrever como as operações de navegação e a operação **Tamanho** são implementadas sobre a sequência de parênteses formada.

Denote por v um nó da árvore original, v' o correspondente de v na árvore intermediária e v_{abre} e v_{fecha} os índices, respectivamente, do abre e do fecha parênteses relacionados a v na sequência. Com isso, as operações suportadas pela representação são descritas de maneira simples.

Se um nó v tem filho esquerdo u, então o primeiro filho de v' é u', logo $v_{\text{abre}} + 1 = u_{\text{abre}}$. De forma semelhante, se v tem um filho direito u, então o irmão seguinte de v' é u', assim $v_{\text{fecha}} + 1 = u_{\text{abre}}$. Alternativamente, se u é pai de v, então $v_{\text{abre}} - 1 = u_{\text{abre}}$ se v é filho esquerdo de u, ou $v_{\text{abre}} - 1 = u_{\text{fecha}}$, caso contrário.

Dessa forma, se o identificador para um nó v na representação é v_{abre}, as operações de navegação podem ser expressas utilizando somente a operação Match. Veja o Programa 3.4.

O que faz com que essa representação se sobressaia quando comparada com a representação de Clark [Cla97] é o suporte à operação **Tamanho**. Para calcular essa, basta notar que os nós contidos na subárvore de v estão distribuídos nas subárvores de v' e de seus irmãos à direita. Dessa forma, se u' é o pai de v', então o tamanho da subárvore de v é $(u_{\text{fecha}} - v_{\text{abre}})/2$, uma vez que há dois parênteses na sequência para cada nó da subárvore de v. Veja a Figura 3.4.

Para encontrar u_{fecha} eficientemente é utilizada a operação Enclose. Veja o Programa 3.5.

Programa 3.4 Navegação na representação de Munro e Raman

```
1   função TemFilhoEsquerdo(id)
2       devolva v[id + 1]                          ▷ é um abre

3   função TemFilhoDireito(id)
4       devolva v[Match(id) + 1]                   ▷ é um abre

5   função FilhoEsquerdo(id)
6       devolva id + 1

7   função FilhoDireito(id)
8       devolva Match(id) + 1

9   função Pai(id)
10      se v[id − 1]                               ▷ é um abre
11          devolva id − 1
12      senão
13          devolva Match(id − 1)
```

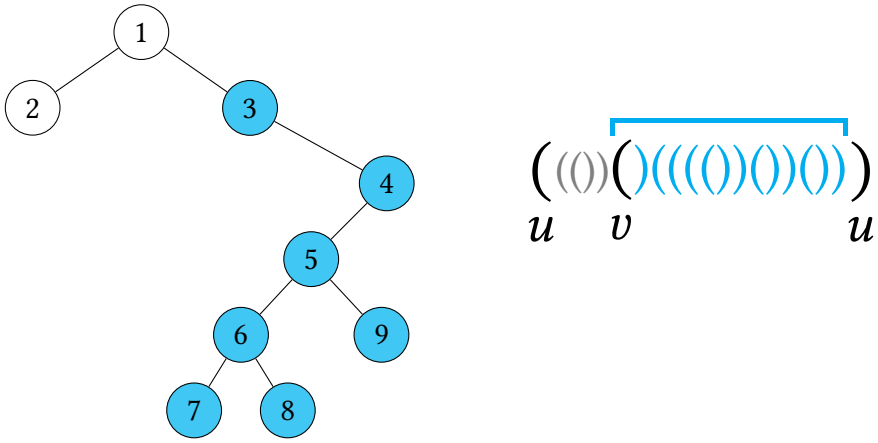

Figura 3.4: *Parênteses contados pela operação* ***Tamanho*** *e sua correlação com os nós da árvore.*

Como as operações Match e Enclose são suportadas sucintamente pela estrutura de Geary, Rahman, Raman e Raman [Gea+06], tem-se que a implementação resultante da árvore binária pode ser mantida em espaço $2n + o(n)$, ou seja, sucintamente.

3.2.1 Estendendo a representação de Munro e Raman

Note que representação as folhas da árvore original formam subsequências da forma "())". Com uma estrutura sucinta que dê suporte às operações $\text{Rank}_{())}$ e $\text{Select}_{())}$ que, em vez de buscarem por bits 1 como as operações comuns, buscam por esse padrão, é possível disponibilizar implementações sucintas e eficientes para operações que envolvem folhas.

De fato, é possível estender a estrutura do Capítulo 1 para permitir Rank e Select de qualquer padrão de bits de tamanho p constante mantendo sua eficiência de custo

Programa 3.5 Operação **Tamanho** na representação de Munro e Raman

```
1  função Tamanho(id)
2      uAbre ← Enclose(id)
3      uFecha ← Match(uAbre)
4      devolva (uFecha – id) / 2
```

de execução e de memória. Essa extensão se dá ao substituir, desde a construção até a utilização, as leituras de trechos dos vetores de bits de tamanho k por leituras de trechos de tamanho $k+p$ e substituindo testes que verificam se um bit é 1 por outros que verificam se o trecho lido se assemelha ao padrão. Note que essas alterações fazem com que os algoritmos comuns sejam executados em um vetor virtual (já que esse não pode ser armazenado sucintamente), onde seu i-ésimo bit é 1 se no trecho entre os bits i e $i + p$ da sequência original ocorre uma instância do padrão.

Com isso, é possível implementar as seguintes operações:

- **RankFolha** recebe o identificador de um nó e devolve quantas folhas da árvore original estão à esquerda do nó, ou seja, quantas folhas seriam visitadas até visitar esse nó em um percurso in-ordem.
- **SelectFolha** recebe um número i e devolve o identificador do i-ésima folha visitada em um percurso in-ordem.
- **ContagemDeFolhas** recebe o indentificador de um nó v e devolve a número de folhas da subárvore de v.

As duas primeiras operações decorrem diretamente da ordem em que as folhas aparecem na sequência. Note que os fecha parênteses da sequência ocorrem na mesma sequência em que os vértices são visitados em um percurso in-ordem. Reveja as Figuras 3.2 e 3.3. Dessa forma, é possível implementar eficientemente **RankFolha** e **SelectFolha** utilizando $\textsc{Rank}_{())}$ e $\textsc{Select}_{())}$, conforme as duas primeiras rotinas do Programa 3.6.

Para saber o número de folhas da subárvore de v, semelhante à operação **Tamanho**, é necessário encontrar u_{fecha}, tal que u' é pai de v'. Como todos os parênteses que representam vértices da subárvore de v estão entre v_{abre} e u_{fecha}, então o número de folhas da subárvore de v é igual ao número folhas que aparecem na sequência entre v_{abre} e u_{fecha}, o que pode ser obtido com $\textsc{Rank}_{())}$ com custo constante. Veja a última rotina do Programa 3.6.

Programa 3.6 Operação sobre as folhas na representação de Munro e Raman

```
1  função RankFolha(id)
2      devolva Rank_())(Match(id))

3  função SelectFolha(i)
4      devolva Select_())(i)

5  função ContagemDeFolhas(id)
6      uFecha ← Match(Enclose(id))
7      devolva Rank_())(uFecha) – Rank_())(id – 1)
```

Capítulo 4

Conclusão

Inicialmente, planejávamos estudar árvores de sufixos sucintas e *Wavelet Trees*, duas estruturas utilizadas na busca de padrões em textos cujo alfabeto não é binário. É notável que a dificuldade do assunto foi subestimada, uma vez que essas metas não puderam ser alcançadas, devido a impasses no decorrer do projeto.

O maior desses impasses foi encontrado ao implementar a representação de sequências balanceadas de parênteses descrita por Munro e Raman [MR97]. Em específico, não obtivemos uma implementação conclusiva da operação Enclose para pares de parênteses que os autores classificam como *minúsculos*.

Após algumas semanas sem sucesso, buscamos por outras representações sucintas para essas sequências e encontramos a apresentada no Capítulo 2. Essa, apesar de exigir mais embasamento teórico, forma uma implementação que é nitidamente mais robusta e concisa.

A partir das estruturas contempladas por esse texto, cujas implementações feitas durante o projeto podem ser encontradas no GitLab, e algumas outras que foram visitadas com menos profundidade, nota-se que há um padrão entre elas. De fato, todas elas se baseiam em reduzir as estruturas a subestruturas de tamanho $\left\lceil \frac{\lg n}{2} \right\rceil$, nas quais todas as operações possíveis são armazenadas em tabelas de consulta. Analisando melhor essa técnica, vê-se que é necessário somente que a estrutura seja redutível a subestruturas de tamanho $a \lg n$, onde $a \in (0, 1)$. Com isso, cada subestrutura pode ser utilizada como um dos índices de tabelas que contêm todas as respostas locais à subestrutura, uma vez que o conteúdo completo da subestrutura pode ser lido com somente uma chamada à primitiva **LerDeVetorDeBits** e essas tabelas podem ser mantidas em espaço $O\left(n^a \lg^k n\right)$, onde k é uma constante, ou seja, em espaço $o(n)$, já que $a < 1$.

Referências

[Cla97] David Clark. “Compact PAT trees”. Tese de dout. 1997 (ver pp. 2, 4, 8, 27–29, 31).

[Die17] Reinhard Diestel. *Graph Theory*. 5th. Springer Publishing Company, Incorporated, 2017. ISBN: 3662536218 (ver p. 17).

[Emd91] Peter van Emde Boas. “Machine Models and Simulations”. Em: *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. Cambridge, MA, USA: MIT Press, 1991, pp. 1–66. ISBN: 0444880712 (ver pp. 1, 2).

[Gea+06] Richard F. Geary, Naila Rahman, Rajeev Raman e Venkatesh Raman. “A simple optimal representation for balanced parentheses”. Em: *Theoretical Computer Science* 368.3 (2006). Combinatorial Pattern Matching, pp. 231–246. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.09.014 (ver pp. 15–17, 25, 32).

[Jac88] Guy Joseph Jacobson. “Succinct Static Data Structures”. AAI8918056. Tese de dout. USA, 1988 (ver pp. 1, 2, 4, 16).

[MR97] James Ian Munro e Vankatesh Raman. “Succinct representation of balanced parentheses, static trees and planar graphs”. Em: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 1997, pp. 118–126. DOI: 10.1109/SFCS.1997.646100 (ver pp. 27, 30–34).

[RRS07] Rajeev Raman, Venkatesh Raman e Srinivasa Rao Satti. “Succinct Indexable Dictionaries with Applications to Encoding K-Ary Trees, Prefix Sums and Multisets”. Em: *ACM Trans. Algorithms* 3.4 (nov. de 2007), 43–es. ISSN: 1549-6325. DOI: 10.1145/1290672.1290680 (ver pp. 18, 19).

[Sta99] Richard P. Stanley. *Enumerative Combinatorics*. Vol. 2. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1999. DOI: 10.1017/CBO9780511609589 (ver p. 27).