UNIVERSIDADE ESTADUAL DO MARANHÃO
CENTRO DE CIÊNCIAS TECNOLÓGICAS
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO E SISTEMAS
Disciplina: ALGORITMOS E ESTRUTURAS DE DADOS
Professor: Dr. Reinaldo

Resenha do Capítulo 3 de "Estrutura de Dados e Algoritmos" de Michael T. Goodrich e Roberto Tamassia

Gabriele de Sousa Araújo

GOODRICH, Michael T.; TAMASSIA, Roberto. Estruturas de dados e algoritmos em java. Grupo A, 2013. E-book. ISBN 9788582600191. p. 113-154. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788582600191/ . Acesso em: 12 jun. 2024.

O capítulo 3 do livro "Estrutura de Dados e Algoritmos" trata dos conceitos de índices, nodos e recursão em Java, onde são explorados criptografia simples com arranjos de caracteres, listas simples e duplamente encadeadas, listas encadeadas circulares e ordenação de listas encadeadas e recursão. A seguir, serão exploradas as principais seções abordadas entre as páginas 113 e 154, incluindo exemplos e de aplicação.

A seção inicial do capítulo aborda a utilização de arranjos de caracteres, onde objetos strings são armazenados internamente como um arranjo. Em Java, é fácil converter entre strings e arranjos de caracteres: para criar uma string a partir de um arranjo, usa-se ***new*** *String(A)*, e para converter uma string em arranjo, utiliza-se *S.toCharArray()*. Por exemplo, o arranjo [a, c, a, t] se transforma na string "acat", e a string "adog" se transforma no arranjo [a, d, o, g]. Como exemplo da conversão entre strings e arranjos de caracteres na criptografia é a Cifra de César usado por Júlio César para proteger mensagens militares, onde cada letra de uma mensagem é substituída pela letra que está três posições à frente no alfabeto (Figura 01).

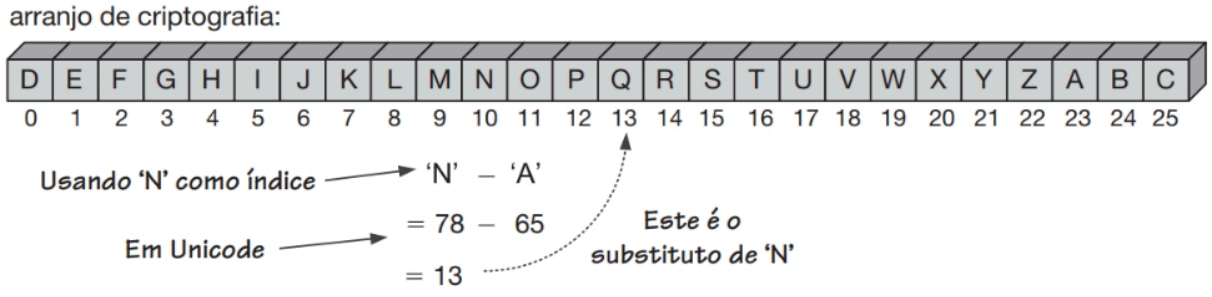

Figura 01. Demonstração do uso de caracteres maiúsculos como índices de arranjos. Nesse caso, para executar a regra de substituição do mecanismo de criptografia da Cifra de César.

Se as letras forem numeradas como índices de um arranjo (A=0, B=1, etc.), a Cifra de César pode ser expressa como: *substitua cada letra i pela letra (i + 3) mod 26*. O operador módulo (% em Java) facilita o giro ao redor do alfabeto. Para decriptar, basta substituir cada

letra por uma três posições antes. Usando arranjos, pode-se criar um arranjo *encrypt* para criptografar e um arranjo *decrypt* para decriptar. Em Java, caracteres são armazenados como números Unicode, permitindo o uso de letras maiúsculas como índices de um arranjo para aplicar a regra de substituição.

A seção seguinte, "Arranjos bidimensionais e jogos de posição", explora como muitos jogos de computador utilizam tabuleiros bidimensionais, assim, é visto que programas que lidam com esses jogos necessitam de uma maneira de representar objetos em um espaço bidimensional, frequentemente usando arranjos de duas dimensões, onde dois índices (i e j) referenciam as células do arranjo. Em Java, pode-se definir um arranjo bidimensional como um arranjo de arranjos, criando matrizes que permitem manter tabuleiros bidimensionais e realizar cálculos com os dados nas linhas e colunas. Por exemplo, int[][] Y = new int[8][10] cria um arranjo com 8 linhas e 10 colunas. Um exemplo é o jogo da velha, em que cada célula da matriz (3×3) armazena o estado do jogo, ou seja, um arranjo bidimensional (*board*).

Por fim, em Java, arranjos bidimensionais são na verdade arranjos de uma dimensão aninhados, levando a questões sobre a composição de objetos. Por exemplo, *deepEquals(A, B)* verifica a igualdade profunda, e *deepToString(A)* converte arranjos bidimensionais em *strings*. No entanto, não há um método *deepCopyOf* para criar cópias idênticas, exigindo o uso de *clone()* ou *copyOf()* para cada linha individualmente.

Como visto, os arranjos são simples para armazenar dados em ordem, mas têm limitações como a necessidade de prever seu tamanho e usar índices para acessar elementos. Uma alternativa são as **listas simplesmente encadeadas**, onde uma coleção de nodos forma uma ordem linear, com cada nodo (objeto) armazenando uma referência para um elemento e outra para o próximo nodo (*next*), eliminando as limitações dos arranjos. Como exemplo, tem-se na Figura 02 uma lista simplesmente encadeada cujos elementos são strings indicando códigos de aeroportos. Os ponteiros *next* de cada nodo são representados como setas. O objeto *null* é denotado como ∅.

Figura 02. Exemplo de lista simplesmente encadeada.

Em uma lista simplesmente encadeada, cada nodo tem uma referência (*next*) para outro nodo, permitindo saltos entre nodos. A lista começa na cabeça (*head*) e termina na cauda (*tail*), identificada por uma referência *next* nula. O código de implementação dessa lista

pode ser observado com mais detalhes na página 122 do livro no *Trecho de Código 3.12*. Conforme os Trechos de Código 3.14 - 3.16 Para inserção de um elemento na cabeça (head) de uma lista simplesmente encadeada, usa-se o algoritmo *addFirst(v)*, que ajusta a referência *next* do novo *nodo* para o antigo *head* e atualiza *head* para o novo nodo. Para inserir na cauda, o algoritmo *addLast(v)* ajusta a referência *next* do antigo *tail* para o novo nodo e atualiza tail. A remoção da cabeça, feita pelo algoritmo *removeFirst()*, atualiza *head* para o próximo nodo e seta *next* do nodo removido para *null*.

Remover um elemento da cauda ou de qualquer posição de uma lista simplesmente encadeada é ineficiente, pois requer percorrer a lista para encontrar o nodo anterior. Nesse caso são exploradas as **listas duplamente encadeadas** que permitem movimento em ambas as direções, solucionam esse problema. Um nodo em uma lista duplamente encadeada armazena duas referências: next (para o próximo nodo) e prev (para o nodo anterior), permitindo inserções e remoções eficientes em ambas as extremidades e no meio. A implementação Java de um nodo de lista duplamente encadeada inclui campos para armazenar o elemento e as referências prev e next, além de métodos para acessar e modificar esses campos.

Para simplificar a programação, listas duplamente encadeadas podem ter sentinelas (*header* e *trailer*) que não armazenam elementos. O *header* tem *next* válido e *prev* nulo, enquanto o trailer tem prev válido e next nulo como demonstrado na Figura 03.

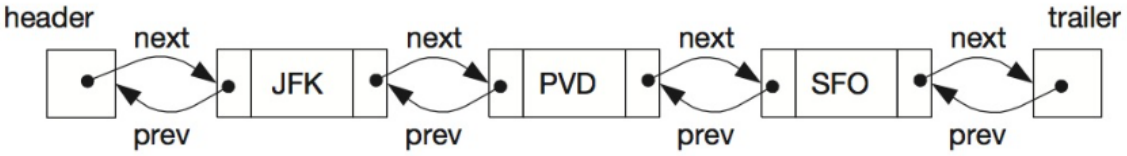

Figura 03. Uma lista duplamente encadeada com sentinelas, *header* e *trailer*, marcando as extremidades da lista.

Essas sentinelas facilitam inserções e remoções em ambas as extremidades da lista. O algoritmo *addFirst(v)* insere um novo nodo no início, ajustando as referências do *header* e do próximo nodo. O algoritmo *removeLast()* remove o último nodo ajustando as referências do trailer e do nodo anterior, ambos apresentados nos Trechos de código 3.18 e 3.19.

Agora referente as inserções e remoções eficientes no meio da lista duplamente encadeada, o algoritmo *addAfter(v, z)* insere um novo nodo z após o nodo v, ajustando as referências *prev* e *next* dos nodos envolvidos, como descrito no Trecho de Código 3.20. A remoção de um nodo do meio é realizada pelo algoritmo *remove(v)*, que desconecta v

ajustando as referências dos nodos adjacentes, como mostrado no Trecho de Código 3.21. A implementação das listas duplamente encadeadas é apresentada no Trecho de Codigo 3.22 - 3.24.

Na seção 3.4 são descritas algumas aplicações e extensões de listas encadeadas. Uma delas são as listas encadeadas circulares, úteis em várias aplicações como jogos de roda. Essas listas não têm início ou fim; o último nodo aponta para o primeiro, e algum nodo é marcado como cursor para percorrer a lista. Seus métodos principais incluem *add(v)*, que insere um nodo após o cursor, *remove()*, que remove o nodo após o cursor, e *advance()*, que move o cursor para o próximo nodo (Trecho de Código 3.25). Como exemplo, a brincadeira "Pato, Pato, Ganso" pode ser simulada usando listas circulares, onde crianças representam nodos e o pegador é identificado e removido da lista, avançando o cursor até um "ganso" ser escolhido, destacando a utilidade dessas listas na simulação de jogos.

Além disso, para ordenar uma lista encadeada, pode-se usar o algoritmo de inserção ordenada, detalhado no Trecho de Código 3.27 onde é visto que o algoritmo *InsertionSort(L)* ordena uma lista duplamente encadeada L com elementos comparáveis em ordem não decrescente e percorre a lista, removendo cada nodo (*pivot*) e inserindo-o na posição correta antes de continuar com o próximo nodo. Conseguinte, a implementação desse algoritmo para uma lista duplamente encadeada é apresentada no Trecho de Código 3.28.

Por fim, na seção 3.5 é visto que as repetições podem ser implementadas usando laços (*for, while*) ou recursão, onde uma função chama a si mesma (chamada recursiva). Um de seus maiores benefícios em projeto de algoritmos é que nos permite tirar vantagem da estrutura repetitiva presente em muitos problemas, alguns dos métodos *deepEquals()* e *deepToString()* descritos anteriormente são definidos recursivamente. Para simplificar o código em alguns casos e como demonstração de recursão, é descrita a implementação recursiva da função fatorial em Java (Trecho de Código 3.29) que substitui a necessidade de laços com chamadas recursivas, simplificando o código em alguns casos. A função fatorial, que multiplica inteiros de 1 a n, pode ser definida recursivamente: factorial(n) = n * factorial(n-1) com o caso base factorial(0) = 1.

A forma mais simples de recursão é a recursão linear, onde um método faz, no máximo, uma chamada recursiva por ativação. Um exemplo é somar elementos de um arranjo, onde a soma de n elementos pode ser definida como a soma dos n-1 primeiros elementos mais o último (Trecho de Código 3.31). Este método, *LinearSum*, garante que o algoritmo termina ao definir um caso base não recursivo para n=1 e chamar recursivamente

para n-1 até atingir o caso base. A recursão linear segue a estrutura típica de testes de casos base seguidos pela chamada recursiva.

Para analisar algoritmos recursivos, usa-se o rastreamento recursivo, criando uma caixa para cada instância do método e visualizando as chamadas recursivas com setas. Por exemplo, o rastreamento do algoritmo *LinearSum* mostra que, para um arranjo de tamanho n, são feitas n chamadas, consumindo tempo e memória proporcional a n. Outro exemplo é o algoritmo *ReverseArray* (Trecho de Código 3.32), que inverte elementos de um arranjo trocando o primeiro com o último e chamando recursivamente para os elementos restantes. Este algoritmo garante a terminação com dois casos base: i = j e i > j.

A recursão binária faz duas chamadas recursivas, como somar elementos de um arranjo dividindo-o em duas metades e somando os resultados. *BinarySum* (Trecho de Código 3.34) é mais eficiente em espaço do que *LinearSum*, pois a profundidade da recursão é proporcional a log2(n). A recursão múltipla ocorre quando um método faz várias chamadas recursivas, usada para enumerar todas as configurações possíveis em problemas combinatórios, como resolver quebra-cabeças de soma. O algoritmo *PuzzleSolve* (Trecho de Código 3.37) exemplifica essa abordagem, gerando e testando todas as permutações de elementos.