Monitoria de Estrutura de Dados

Univesidade Federal do Ceará
Francisco Alex Sousa Anchieta

Sumário

1 Recursão

Este é um conceito que na maioria das vezes nós, alunos, temos dificuldade de compreender. O que é estranho, uma vez que muitos problemas resolvemos quase que imediato com um versão recursiva, ainda que não sabemos explicar o porquê. Por exemplo, considere o fatorial de um número n, definido por

$$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1$$

Observe que $n! = n \cdot (n-1)!$ e que $(n-1)! = (n-1) \cdot (n-2)!$ e assim por diante. Ou seja, para encontrarmos o fatorial de n, podemos chamar os fatoriais de números menores que n e resolvê-los, multiplicamos o seu resultado, até chegarmos em 1, de forma que obtemos o resultado da operação.

Vemos isso, da mesma forma, na exponenciação, onde k^n pode ser definido como $k^n = k \cdot k^{(n-1)}$, sendo $k^{n-1} = k \cdot k^{(n-2)}$ e assim consecutivamente até chegarmos em k^1 e, como resultado, temos uma multiplicação de n k's.

1.1 E o que é uma Recursão?

Um procedimento é dito recursivo se ele é definido em termos de si mesmo, ou seja, é um método de resolução de problemas que envolve quebrar um problema em subproblemas menores até chegar a um problema pequeno o suficiente para que ele possa ser resolvido trivialmente[1].

Como a recursão é definida a partir de versões de si mesma, devemos observar o momento em que as nossas chamadas deve acabar. Dessa forma, precisamos de um caso base para o nosso problema, ou melhor, uma condição de parada que conclua essa sub-rotina. Este estado define o ponto final da chamada recursiva, onde a solução é trivial, por isso chamamos de caso base. E para isso precisamos da condição. Para os números fatoriais, por exemplo, temos que o caso base é 1!, e para a exponenciação, k^1.

Agora, vamos exemplificar a recursão. Para isso utilizarei um pseudo-código, escrito em português, que facilitará o entendimento, pois não se limita as regras de uma linguagem. Observe o pseudo-código que encontra o número fatorial de n:

Algorithm 1 Calcula o Número Fatorial de x com Recursão

```
função FATORIAL(x)
    se x = 1 então
        retorne 1
    senão
        retorne x * FATORIAL(x − 1)
    fim se
fim função
```

Note que a função é, basicamente, um **se** e **senão**. Quando calculamos o fatorial, temos que a condição verifica se a entrada é igual a 1, caso seja, retorna 1 como resultado. Isso ocorre pois sabemos que 1! = 1, sendo este o último valor calculado e, assim, 1! é o caso base e o $x == 1$, a condição de parada. No **senão**, temos uma chamada recursiva que decrementa o valor de entrada x e multiplica essa chamada com o próprio x, isso até chegarmos no caso base.

Importante destacar que um código escrito de forma recursiva podemos escrever o mesmo problema por meio de iterações (o uso laços de repetição), mas isso não quer dizer que seja mais fácies. Normalmente, códigos recursivos são mais legíveis e mais simples de se implementar. Por exemplo, podemos escrever a função anterior de forma iterativa:

Algorithm 2 Calcula o Número Fatorial de x com Iteração

```
1: fat ← 1
2: para i ← 1 até n faça
3:     fat ← fat ∗i
4: fim para
5: retorne fat
```

A forma iterativa do cálculo do número fatorial ainda é simples, porém ainda temos que adicionar uma variável na função. Para evidenciarmos essa diferença veremos a sequência de Fibonacci.

1.2 A recursão na sequência de Fibonacci

A sequência de fibonacci é uma sequência de números inteiros, começando de 1, onde cada termo subsequente corresponde à soma dos dois anteriores. Com isso, temos que os números de fibonacci são os integrantes dessa sequência, sendo:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...$$

Observe que, inicialmente, os valores crescem lentamente, porém, mais adiante, as somas se tornam cada vez maiores, de forma que fique mais complicado computá-las. Por exemplo, temos que $F_{50} = 12586269025$, $F_{55} = 139583862445$ e o $F_{99} = 218922995834555169026$. Veja que para computar F_{99}, utilizar **int** do C, por exemplo, não vai ser possível. Se desejar verificar os valores de fibonacci para valores de n até 100, veja aqui. Se você é mais *hardcore*, veja aqui os 2000 primeiros números de fibonacci.

Podemos representar essa sequência por uma relação de recorrência que seria uma fórmula:

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{(n-1)} + F_{(n-2)}$$

Note que essa fórmula possui uma recursão em seu caso geral, de forma que eu precise de resultado anteriores para resolver o atual.

Vamos aplicar o fibonacci em nosso pseudo-código na forma recursiva:

Algorithm 3 Calcula o Número Fibonacci na posição n com Recursão

```
função FIBONACCI(n)
    se n < 2 então
        retorne n
    senão
        retorne FIBONACCI(n − 1) + FIBONACCI(n − 2)
    fim se
fim função
```

Perceba que essa função é tão simples quanto o do número fatorial, pois temos apenas uma condicional que direciona o que deve ser feito. A condição de parada resolve os dois casos base, onde para $n = 2$, temos que $F_2 = 2 = n$. A chamada recursiva no **senão** é apenas a aplicação da fórmula apresentada anteriormente. Isso evidencia a natureza recursiva desse problema.

Agora, observe a versão iterativa:

Algorithm 4 Calcula o Número Fibonacci na posição n com Iteração

```
1: j ← 1
2: i ← 1
3: para k ← 1 até n faça
4:     t ← i + j
5:     i ← j
6:     j ← t
7: fim para
8: retorne j
```

Note que a forma iterativa dessa função não é nem um pouco legível quanto a versão recursiva. Nela, se utiliza três variáveis auxiliares para realizar as somas. A variável i representa $F_{(n-2)}$ e j representa $F_{(n-1)}$ no laço de repetição. Quando $k = n$, a funcão retorna j pois, na iteração anterior, $j \leftarrow t = F_{(n-2)} + F_{(n-1)}$.

Além dos números de fibonacci, existe outros problemas que o uso de recursão é evidenciável. Como exemplo, temos estruturas de dados dinâmicas, como as Árvores, e o método de ordenação QuickSort (estes são temas que vão ser estudados nessa disciplina, futuramente). Além disso, quando utilizamos uma linguagem funcional, é mais prático o uso de recursões (para quem faz Ciência da Computação, vai compreender melhor futuramente, para os outros, vale a dica!).

1.3 O seu lado negativo

Ainda que o uso de recursão seja mais legível e a solução, para alguns problemas, seja imediato, devemos ter cautela ao aplicarmos na maioria dos casos. No exemplo anterior, o uso da versão iterativa do Fibonacci é mais eficiente do que a recursiva, de forma que a versão recursiva tem um custo exponencial (inicialmente confie em mim, isso é ruim!) e a iterativa, um custo linear.

Outra desvantagem do uso de recursão é seu alto consumo de memória, uma vez que é necessário armazenar o estado da chamada anterior até que o caso base ocorra. Por isso, devemos ter cuidado ao usar esse recurso em nosso programas. O ideal é implementar a versão iterativa no código final e utilizar a recursiva apenas pra entendimento e legibilidade.

2 Análise de Algoritmos

2.1 O que são Algoritmos?

Os algoritmos fazem parte do nosso dia-a-dia, ainda que não percebemos, sendo de grande importância. Por exemplo, quando compramos um móvel, precisamos realizar a sua montagem e, para tal, é necessário um guia do produto. Caso não houvesse este guia, provavelmente não teremos um móvel útil. No entanto, com o guia em mãos, temos todos os requisitos (ou quase) para efetuarmos a montagem.

Bem, e o que este guia descreve? Ele define os **passos** que deve ser seguidos para a construção ideal do móvel, ou melhor, descreve como deve ser montado e onde cada peça deve ser encaixada. Com isso, temos que o guia é o **algoritmo** que devemos usar para a montagem de um móvel qualquer.

Agora, vamos escrever o passo a passo para realizarmos a travessia de uma rodovia:

1. Olhar para o lado esquerdo
2. Olhar para o lado direito
    a) Se não vem veículos na sua direção, atravesse
    b) Senão, não cruze a rodovia

Obviamente, temos outras formas de realizar essa travessia (poderíamos executar uma verificação depois de olharmos para esquerda e outra ao olharmos para a direita) e, ainda assim, conseguimos satisfazer o nosso objetivo. Dessa maneira, um mesmo problema pode ser resolvidos de diversas formas. Porém, devemos observar se as soluções são aplicáveis e corretas.

Dado exemplos textuais de algoritmos, temos uma ideia de como ele funciona. Dito isso, agora vamos definir o que é um algoritmo na computação. Um algoritmo é uma sequência de passos computacionais que transformam a entrada na saída[2]. Em outras palavras, é uma serie finita de passos que levam à execução de uma tarefa. Quando programamos `print("Hello World")`, estamos ordenando que seja apresentado no console a mensagem *Hello World*.

Desse modo, todas as tarefas executadas pelo computador são baseadas em algoritmos. Isso ocorre porque devemos dizer a ele exatamente o que queremos que ele faça (e ele é bastante obediente!). Portanto, quando o nosso código possui um erro, por exemplo, a culpa sempre é do programador, pois a máquina só faz o que mandamos ela fazer.

2.2 Algoritmo de Euclides

A partir de diante, vamos lidar com um dos exemplos clássicos de algoritmo numérico, o algoritmo de Euclides. Euclides de Alexandria foi um grande matemático da Grécia Antiga e, entre tantos outros estudos, ele desenvolveu um método para calcular o máximo divisor comum (MDC) entre dois números inteiros não nulos. O MDC de dois números inteiros é o maior número inteiro que divide ambos sem deixar resto. O algoritmo utiliza a recursão como sua aliada (Vê Seção 1). Ele utiliza o resto da divisão como entrada para a chamada recursiva, ou seja, MDC(a, b) $\Rightarrow$ MDC(b, r), onde $a, b \in \mathbb{Z}_*$ e $r = a \ mod \ b$. A condição de parada é $b = 0$.

Os passos que devemos seguir são:

1. Calcule $r = a \ mod \ b$
2. Se $r = 0$, o MDC de a e b é a
3. Senão, o MDC de a e b é MDC(b, $a \ mod \ b$)

Por exemplo, considere o MDC(102, 80). Temos:

$$\text{MDC}(102, 80) \Rightarrow \text{MDC}(80, 22), \text{ pois } 102 \bmod 80 = 22 \quad (1)$$
$$\text{MDC}(80, 22) \Rightarrow \text{MDC}(22, 14), \text{ pois } 80 \bmod 22 = 14 \quad (2)$$
$$\text{MDC}(22, 14) \Rightarrow \text{MDC}(14, 8), \text{ pois } 22 \bmod 14 = 8 \quad (3)$$
$$\text{MDC}(14, 8) \Rightarrow \text{MDC}(8, 6), \text{ pois } 14 \bmod 8 = 6 \quad (4)$$
$$\text{MDC}(8, 6) \Rightarrow \text{MDC}(6, 2), \text{ pois } 8 \bmod 6 = 2 \quad (5)$$
$$\text{MDC}(6, 2) \Rightarrow \text{MDC}(2, 0), \text{ pois } 6 \bmod 2 = 0 \quad (6)$$

Como $a = 2$, o MDC(102, 80) = 2.

Essa descrição textual fornece os passos que devemos seguir. Não obstante, podemos escrever em uma forma que o computador possa entender. E não é difícil, porque esta é a forma que damos ordens ao computador. Vamos utilizar as linguagens de programação para resolver esse problema em um computador. Segue o pseudo-código:

Algorithm 5 Calcula o MDC entre dois inteiros não nulos, pelo método de Euclides

```
função MDC(a, b)
    se b = 0 então
        retorne a
    senão
        retorne MDC(b, a mod b)
    fim se
fim função
```

2.3 O conceito da análise algorítmica

Sabemos o que é um algoritmo e o porque ele é tão importante no nosso estudo. Agora, podemos lidar com sua complexidade e o quanto de recurso ele consome. Essas informações são de extrema importância uma vez que podemos descobrir a sua correção e desempenho.

Ainda que os computadores modernos sejam extremamente rápidos e precisos, eles possuem uma limitações. Por exemplo, ele não consegue representar todos os números inteiros, pois são infinitos, e muito menos os números reais. Por isso, operações que lidam com números reais (como algumas divisões, cálculo de derivadas, etc) precisam de uma atenção melhor.

Como dito anteriormente, conseguimos resolver um problema de várias formas possíveis, porém uma escolha errada pode fazer com que o computador não faça o que se pede em um tempo hábil. Por essa razão, realizarmos a análise de algoritmos é de suma importância, ela tem como função determinar os recursos necessários para executar um dado algoritmo. Dessa forma, dados dois algoritmos para um mesmo problema, a análise permite decidir qual dos dois é mais eficiente

Por exemplo, pouco tempo atrás lidamos com os números de *fibonacci*, e foi apresentado dois algoritmos que realizam a mesma operação: um utilizava a recursão, com um código mais legível; e outro na forma iterativa, um pouco mais difícil de compreender. Foi dito ainda, que a forma iterativa é mais eficiente. Isso ocorre pois cada chamada recursiva chama mais duas em seu escopo, de forma que acumula diversas outras chamadas, que são até redundante. Observe que, na primeira chamada, cria-se outras 2; na segunda, é criado 4; na terceira, 8; e na n-ésima, 2^n chamadas. Sendo assim, temos que a quantidade de operações realizadas no versão recursiva cresce exponencialmente e, com isso, se precisarmos de 10 chamadas, teremos $2^{10} = 1024$ procedimentos para resolver.

Em contra-partida, a versão iterativa possui um crescimento linear. Ela possui um *loop* que realiza 4 operações (uma soma e três atribuições) que são realizadas n-1 vezes, de modo que efetua 4n-4 operações. Com as duas atribuições antes do laço, temos 4n-2 operações. À vista disso, a versão iterativa cresce segundo uma função afim crescente. Adiante, explicaremos o que essas operações significam.

Para realizarmos a análise, precisamos ir além do resultado final do algoritmo, devemos saber o que é seu tempo de execução. Para isso, sempre expressaremos o tempo de execução contando o número de etapas básicas do computador, em função do tamanho da entrada[3]. As operações básicas são as unidade. Dessa forma, atribuições, operações aritméticas básicas, comparações e acesso de elementos em uma estrutura de dados simples são ditos passos básicos. Essas são as unidades de um algoritmo. Observe que o tempo de execução representa uma função, não necessariamente é uma constante, uma vez que nos baseamos na entrada.

Quando lidamos com o fibonacci iterativo, foi dito que ele realiza $4n-2$ instruções. Esse é o tempo de execução desse algoritmo. Normalmente, usaremos esse custo para representar a complexidade do algoritmo.

No entanto, é importante salientar que o tempo de execução e o espaço de memória depende da máquina que usamos, não adianta realizar uma ordenação de 100.000 números em um computador moderno e no ENIAC (primeiro computador criado) esperando o mesmo número de instruções. Disso, temos que:

> "O tempo gasto por uma dessas etapas [ou seja, as instruções básicas] depende crucialmente do processador e até de detalhes como a estratégia de armazenamento em cache (como resultado, o tempo de execução pode diferir sutilmente de uma execução para a outra). A contabilização dessas minúcias específicas da arquitetura é uma tarefa incrivelmente complexa e produz um resultado não generalista de um computador para o outro. Portanto, faz mais sentido procurar uma caracterização organizada e independente de máquina da eficiência de um algoritmo". Dasgupta, Papadimitriou e Vazirani, 2006 [3].

Além de desconsiderarmos a arquitetura da máquina, podemos simplificar (ainda mais) nossa notação lidando apenas com os termos mais significativos. Por exemplo, em vez de representar 4n-2 como o tempo de execução do fibonacci iterativo, podemos definir $O(n)$, uma vez que o coeficiente 4 e a constante 2, para entradas excessivamente grandes, tornam-se insignificantes. Vamos definir o que significa $O(n)$.

2.4 Notação assintótica

A notação assintótica, na análise de algoritmos, é a forma de representarmos o tempo de execução de um algoritmo. Com ela, estamos preocupados com a maneira como o tempo de execução de um algoritmo aumenta, com o tamanho da entrada no limite,

à medida que o tamanho da entrada aumenta indefinitivamente[2]. Isso quer dizer que estamos preocupados com valores grandes de entrada para o processamento do algoritmo, com o intuito de calcular o tempo total de processamento e viabilidade para determinados casos.

Na verdade, a notação assintótica representa classes de funções. Um conjunto de funções estão na mesma classe quando possuem o mesmo termo dominante. Por exemplo, as funções

$$2n^2,\ 5n^2+5,\ 8n^2+5n+8,\ \frac{5}{2}n^2+1526n,\ 0.5n^2+\frac{5}{3},\ \sqrt{5}n^2+5n$$

estão na mesma classe.

Existem diversas notações que relacionam funções, mas iremos introduzir apenas a notação O (lê-se *big-O*). A notação O incorpora as funções que são limitadas superiormente por uma outra função. Denotamos:

Definição 1. $O(g(n)) = \{f(n) | \exists c, n_0 \in \mathbb{R}_+ : 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0\}$

Em outras palavras, existe um número positivo c e um número n_0 tais que $f(n) \leq c \cdot g(n)$ para todo n maior que n_0.

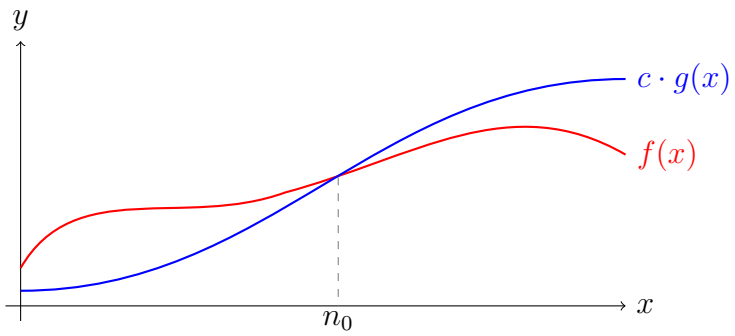

Figura 1 – No ponto n_0, $c \cdot g(x)$ domina $f(x)$, dessa forma $f(x) \in O(g(x))$.

Dessa forma, quando escrevemos $f(n) \in O(g(n))$, dizemos que $f(n)$ é limitada superiormente por $g(n)$, ou que $f(n)$ é dominada por $g(n)$. Dizemos que a notação O fornece limites superiores assintóticos.

Como exemplo, temos que $5n^2+7n-3 \in O(n^2)$, pois $5n^2+7n-3 \leq c \cdot n^2$ quando $c = 6$ e $n_0 = 1$.

Na disciplina de Estrutura de Dados, vamos nos limitar apenas nesse conceito. Na disciplina de Projeto e Análise de Algoritmos, veremos esse assunto mais profundamente (ao menos aos de Ciência da Computação).

Observe que essa análise é puramente matemática, ou seja, independe da linguagem ou máquina, é universal. Em uma análise empírica, a linguagem de programação importa,

além da máquina que usamos. Logo, quando comparamos algoritmos rodando todos no mesmo computador, para verificar o mais rápido, estamos fazendo uma análise empírica. Porém, um dos possíveis problemas em se comparar algoritmos empiricamente é que uma implementação pode ser mais otimizada do que a outra, dependendo da linguagem, da máquina, do compilador e do sistema.

3 Listas encadeadas

Quando estávamos estudando os conceitos fundamentais da programação, nos foi apresentado as principais estruturas e operações que as linguagens possuem. Entre esses diversos conceitos, nos foi apresentado uma estrutura de dados básica, que agrupa os dados, de mesmo tipo, organizando-os de forma contínua na memória. Essa estrutura denominamos vetor, ou ainda, de *array*.

Em um vetor v com n posições, os dados são indexados de 0 à $n-1$ e acessamos o valor por meio de sua posição. Se quisermos o valor da k-ésima posição basta acessarmos v[k], que obteríamos o resultado. Dessa forma, temos uma melhor organização dos dados armazenados do nosso código.

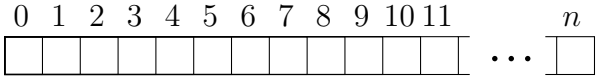

Figura 2 – Estrutura do vetor v na memória

Todavia, existe casos que seu uso não é recomendado. Como os dados são organizados continuamente na memória, o uso de vetores pode causar desperdício. Por exemplo, se criarmos um vetor com 100 posições e usarmos apenas 10, existe outras 90 posições que estão sendo armazenadas inutilmente. Este caso ocorre quando não sabemos a quantidade de dados que precisamos armazenar e acabamos subutilizando-o.

E, relacionado a isso, existe outro problema: quando precisamos de mais posições do que definimos para o vetor. Para resolver isso, devemos alocar mais espaço contínuo à frente. Seria uma solução mas, se este espaço já estar ocupado? Podemos mover todo o vetor para outro espaço na memória que tenha o tamanho que precisamos. Porém, existe outros problemas. Primeiro, todas as vezes que precisarmos alocar mais posições do vetor, teríamos que mover todo o vetor, o que pode ser custoso. E ainda mais: e se não houver mais espaço?

Note que esse problema do uso dos vetores é relativo a forma em que os dados estão sendo armazenados (o que em alguns casos pode ser uma vantagem, uma vez que o acesso dos dados é constante). Devemos então, pensar em uma outra forma de organizar os dados sem eles necessariamente estarem juntos, mas ainda assim interligados. Isso pode parece

contraditório, como vamos manter os dados unidos, mas permitir encontre-se separados? Na verdade, eles estão em endereços da memória não obrigatoriamente consecutivos, mas aponta para o seu próximo, por meio de um ponteiro. Dessa forma, podemos localizá-los e mantê-los conexos.

Para ilustrar isso, imagine que estamos na cidade X e queremos ir para a cidade Y. Para isso, pegamos um ônibus intermunicipal que nos leva ao nosso destino. Entre essas duas cidades, existe k paradas para embarque e desembarque de passageiros distribuídas em k cidades, mas entre os dois destinos pode existir mais cidades.

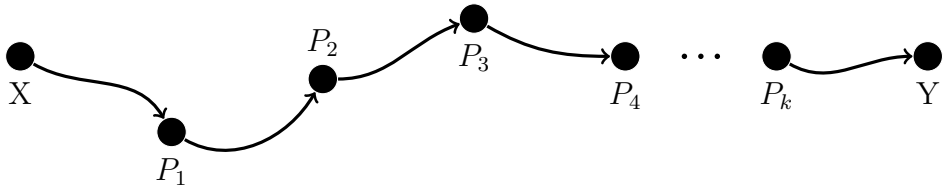

Figura 3 – Rota da cidade X para a cidade Y. Note que devemos passar necessariamente por cada parada para chegarmos ao destino. Ou seja, não podemos pular nenhum ponto.

Como ilustrado na Figura 3, cada ponto possui a rota da parada subsequente, de forma que, de X para Y, passamos por todas as paradas.

Agora, vamos relacionar o nosso exemplo com a solução que buscamos para os vetores. As cidades são as posições na memória, onde cidade vizinhas são os endereços subsequentes. Os pontos de ônibus são as cidades que devemos passar e que temos acesso. Estas cidades são os dados da nossa estrutura.

Dessa forma, podemos definir uma estrutura que armazena o dado propriamente dito e um endereço para o próximo elemento. Esse conjunto dado/endereço chamamos de nó ou célula. Por fim, nomeamos essa estrutura como **lista encadeada**.

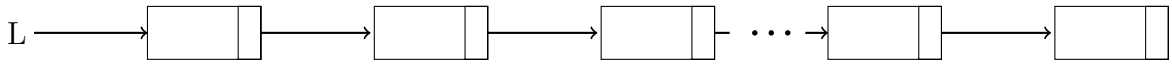

Figura 4 – Ilustração de uma lista encadeada. O endereço da lista é representada por um ponteiro que aponta para o primeiro elemento.

Note que para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado[4].

Como dito anteriormente, a lista aponta para o primeiro elemento, o segundo aponta para o terceiro, o terceiro para o quarto e assim por diante. E o último? Ele aponta para alguém? Na teoria, o último elemento não aponta para ninguém, ou melhor, aponta para um endereço nulo. Nas principais linguagens de programação esse valor de endereço é descrito a partir da palavra-chave *NULL*.

3.1 Operações em Listas encadeadas

Agora, vamos implementar as principais operações em listas encadeadas. Para isso, vamos utilizar uma lista que armazena caracteres. Será utilizado a linguagem C na implementação, uma vez que a sua notação de *struct* é mais simples e estamos mais familiarizado.

Vamos definir a estrutura **celula**:

```
1 struct celula {
2    char caracter;
3    struct cedula *prox;
4 };
5 typedef struct cedlula Celula;
```

Essa estrutura representa a célula da lista, ou seja, cada elemento que pode ser incorporado.

3.1.1 Função de Inicialização

Para inicializarmos a estrutura, devemos criar uma lista vazia. Como a lista aponta para o primeiro elemento e, inicialmente, a lista está vazia, ele apontará para nulo.

```
1 Celula* inicializar() {
2    return NULL;
3 }
```

3.1.2 Função de inserção de Elemento

Com a lista inicializada, podemos adicionar elementos com a função de inserção. Para isso, é criado uma nó auxiliar para armazenar o novo valor. Esse novo elemento passa a apontar para o primeiro elemento e se torna o endereço da lista. Sendo assim, a inserção é dada apenas no início da lista.

```
1 Celula* inserir(char valor, Celula *no) {
2    Celula *nw = (Celula*) malloc(sizeof(Celula));
3    nw->caracter = valor;
4    nw->prox = no;
5    return nw;
6 }
```

E se desejarmos inserir o elemento no meio da lista? Ou no final? Para isso existe outras implementações. Para inserir no final da lista devemos percorrer-la até chegar no último elemento e este deve apontar para o novo elemento criado. Para inserir um elemento na k-ésima posição, devemos percorrer a lista até a posição k definir como seu

sucessor o k-ésimo nó, e o próximo do antigo (k-1)-ésimo elemento o nó adicionado. De toda forma, vamos nos focar apenas na implementação que insere no início.

A figura a seguir demonstra como é realizada essa inserção:

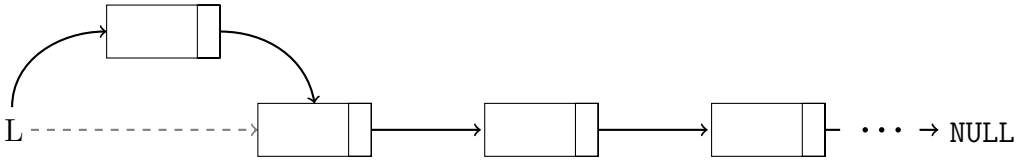

Figura 5 – Inserção de um elemento.

3.1.3 Imprimir Lista

Para sabermos que a nossa lista está funcionando, o ideal é imprimirmos na tela os valores da célula. Como opção, podemos acessar cada posição e verificar o seu valor, utilizando *l->prox->caracter*, *l->prox->prox->caracter*, e assim por diante. Bem, mas não seria a forma mais elegante! Em vez disso, a partir de um loop, imprimimos todos os elementos, percorrendo a lista utilizando uma lista auxiliar. Essa lista é apenas uma cópia e percorremos ela até assumir valor *NULL*.

```
1 void imprimir(Celula *ls) {
2    Celula *p;
3    p = ls;
4    while (p != NULL) {
5       printf("%c\n", p->caracter);
6       p = p->prox;
7    }
8 }
```

3.1.4 Função de busca

A busca segue o mesmo príncipio da função **imprimir**. A diferença é que em **imprimir** é percorrido toda a lista, e na função **busca** percorre somente enquanto não acharmos o valor desejado. Caso encontre o valor, retornamos o endereço do nó em questão. Se não encontramos (quando a lista chega no fim), retornamos *NULL*.

```
1 Lista* busca (Lista* l, char v){
2    Lista* p;
3    while(p != NULL) {
4       if (p->caracter == v) return p;
5       p = p->prox;
6    }
7    return NULL;
8 }
```

3.1.5 Função de remoção

A remoção é um pouco mais difícil que as anteriores, uma vez que existe mais casos para tratarmos. Se queremos retirar o primeiro elemento da lista, basta tornar o segundo nó o endereço da lista. Se o elemento que deve ser removido estar em outra posição, vamos precisar do elemento anterior a ele. Isso é necessário porque precisaremos conectar o seu antecessor com o seu sucessor. A figura a seguir ilustra essa operação:

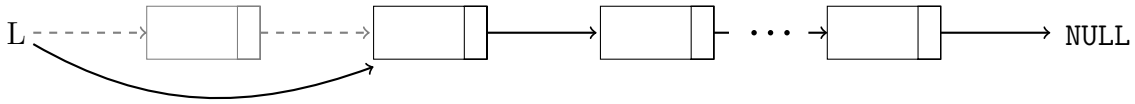

Figura 6 – Remoção do primeiro elemento.

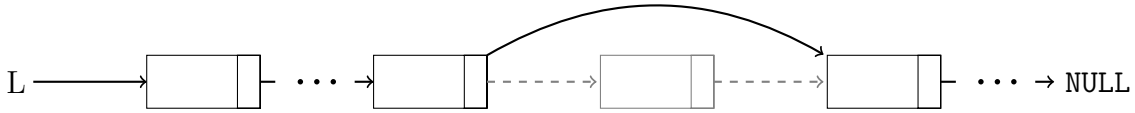

Figura 7 – Remoção do k-ésimo elemento.

Antes disso, precisamos fazer uma busca interna na lista para acharmos a posição que queremos remover e o seu antecessor. O parâmetro da nossa busca é o valor do nó. Caso a busca não encontre nenhuma célula, indica que o elemento não estar na lista e retornamos a lista original. Caso encontre um elemento, aplicamos o caso que a satisfaz, como descrito anteriormente.

```
1  Elemento* remover(Elemento* lst, char valor)
2  {
3     Elemento* anterior = NULL;
4     Elemento* p = lst;
5     while (p != NULL && p->caracter != valor) {
6        anterior = p;
7        p = p->prox;
8     }
9
10    if (p == NULL)
11       return lst;
12
13
14    if (anterior == NULL)
15       lst = p->prox;
16    else
17       anterior->prox = p->prox;
18 }
```

3.2 Vantagens e desvantagens das listas encadeadas

Quando utilizamos listas encadeadas, temos a oportunidade de ser mais livre ao adicionar elementos, pois não nos preocupamos com um limite pré-estabelecido. Além disso, a remoção ou a inserção de elementos não interfere os outros elementos. Mas (como tudo na vida), existe alguns pontos negativos.

Ao utilizar uma lista encadeada, criamos uma dependência do nó com seu antecessor. Isso ocorre pois caso um nó em uma posição k qualquer perca, de alguma forma, o endereço do próximo elemento, todos os $k+1$ elementos serão perdidos. Além disso, para acessar o k-ésimo elemento da lista, temos que percorrer $k-1$ elementos para que, enfim, termos o elemento que desejamos. Se uma lista tem 1000 células e buscamos o elemento da célula da posição 900 temos que percorrer 899 células, enquanto que esse acesso no vetor é constante.

Existe variações da lista encadeada que partem do mesmo princípio, como por exemplo, listas circulares e as listas duplamente encadeadas. Essas veremos mais adiante.

Referências

1 MILLER, B.; RANUM, D. Como pensar como um cientista da computação. 2012. GNU free documentation Licence. Disponível em: <https://panda.ime.usp.br/pensepy/static/pensepy/index.html>. Acesso em: 20 abr. 2020.

2 CORMEN, T. H. et al. Algoritmos: teoria e prática. *Editora Campus*, v. 2, 2002.

3 DASGUPTA, S.; PAPADIMITRIOU, C. H.; VAZIRANI, U. Algorithms. McGraw-Hill, Inc., 2006.

4 CELES, W.; RANGEL, J. L. Caderno didático de estrutura de dados (puc-rio). *Cadernos Pronatec Goiás*, v. 1, n. 1, p. 949–1151, 2018. Disponível em: <http://www.ead.go.gov.br/cadernos/index.php/CDP/article/download/285/210>.