

Monitoria de Estrutura de Dados

Univesidade Federal do Ceará

Francisco Alex Sousa Anchieta

Sumário

1	Recursão	1
1.1	E o que é uma Recursão?	1
1.2	A recursão na sequência de Fibonacci	2
1.3	O seu lado negativo	4
2	Análise de Algoritmos	5
2.1	O que são Algoritmos?	5
2.2	Algoritmo de Euclides	6
2.3	O conceito da análise algorítmica	7
2.4	Notação assintótica	8
3	Listas encadeadas	10
3.1	Operações em listas encadeadas	11
3.1.1	Função de Inicialização	12
3.1.2	Função de inserção de elemento	12
3.1.3	Imprimir lista	13
3.1.4	Função de busca	14
3.1.5	Função de remoção	14
3.2	Vantagens e desvantagens das listas encadeadas	15
3.3	Listas circulares	16
3.4	Listas duplamente encadeadas	18
3.5	DESAFIO: Implementação da BIGINT	19
4	Pilha	20
4.1	Implementação da pilha e suas operações	21
4.1.1	Implementação com um array	21

4.1.2	Implementação com um lista	23
4.2	DESAFIO: Balanceamento de Parênteses	25
5	Filas	25
5.1	Representação e implementação da fila	26
5.1.1	Fila implementada em um array	26
5.1.2	Fila implementada em um lista	29
	REFERÊNCIAS	33

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

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

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 :

Algoritmo 1 Calcula o Número Fatorial de x com Recursão

```
função FATORIAL(x)
  se  $x = 1$  então
    retorne 1
  senão
    retorne  $x * \text{FATORIAL}(x - 1)$ 
  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 fácil. Normalmente, códigos recursivos são mais legíveis e mais simples de se implementar. Por exemplo, podemos escrever a função anterior de forma iterativa:

Algoritmo 2 Calcula o Número Fatorial de x com Iteração

```
1: fat  $\leftarrow$  1
2: para  $i \leftarrow 1$  até  $n$  faça
3:   fat  $\leftarrow$  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:

Algoritmo 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:

Algoritmo 4 Calcula o Número Fibonacci na posição n com Iteração

```

1:  $j \leftarrow 1$ 
2:  $i \leftarrow 1$ 
3: para  $k \leftarrow 1$  até  $n$  faça
4:    $t \leftarrow i + j$ 
5:    $i \leftarrow j$ 
6:    $j \leftarrow t$ 
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 função 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 nossos programas. O ideal é implementar a versão iterativa no código final e utilizar a recursiva apenas pra entendimento e legibilidade.

Exercício 1. *O triângulo de Pascal é um triângulo aritmético infinito onde são dispostos os coeficientes das expansões binomiais. Ele disposto da seguinte forma:*

$$\begin{array}{ccccccc}
 & & \binom{0}{0} & & & & 1 \\
 & & \binom{1}{0} \binom{1}{1} & & & & 1 \quad 1 \\
 & & \binom{2}{0} \binom{2}{1} \binom{2}{2} & & & & 1 \quad 2 \quad 1 \\
 & & \binom{3}{0} \binom{3}{1} \binom{3}{2} \binom{3}{3} & & & & 1 \quad 3 \quad 3 \quad 1 \\
 & & \binom{4}{0} \binom{4}{1} \binom{4}{2} \binom{4}{3} \binom{4}{4} & & & & 1 \quad 4 \quad 6 \quad 4 \quad 1 \\
 & & \dots & & & & \dots
 \end{array}$$

Figura 1 – Representação do triângulo de Pascal

Implemente a função recursiva do triângulo de Pascal.

Exercício 2. *O superfatorial de um inteiro positivo n é dado pelo produto dos n primeiros fatoriais. Ou seja,*

$$sf(n) = \prod_{k=1}^{n-1} k!$$

.

Implemente uma função recursiva para calcular o superfatorial de n .

Exercício 3. *O Hiperfatorial de um inteiro positivo n é definido por:*

$$sf(n) = \prod_{k=1}^n k^k$$

Implemente uma função recursiva que calcula o hiperfatorial

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 resolvido 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 série 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, $\text{MDC}(a, b) \Rightarrow \text{MDC}(b, r)$, onde $a, b \in \mathbb{Z}_*$ e $r = a \bmod b$. A condição de parada é $b = 0$.

Os passos que devemos seguir são:

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

Por exemplo, considere o $\text{MDC}(102, 80)$. 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 $\text{MDC}(102, 80) = 2$.

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:

Algoritmo 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 algumas 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 na 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 unidades. 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 .

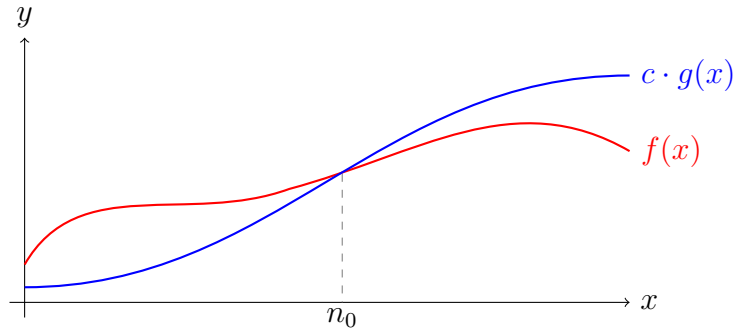


Figura 2 – No ponto n_0 , $c \cdot g(x)$ domina $f(x)$, dessa forma $f(x) \in O(g(x))$.

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.

Exercício 4. Em cada um dos itens, verifique se $f(n) = O(g(n))$.

(a) $f(n) = n \log n$ e $g(n) = 10n \log 10n$

(b) $f(n) = 2^n$ e $g(n) = 2^{n+1}$

(c) $f(n) = n!$ e $g(n) = 2^n$

(d) $f(n) = \sqrt{n}$ e $g(n) = n^{\frac{1}{5}}$

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.

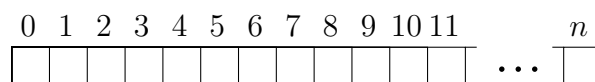


Figura 3 – Estrutura do vetor v na memória

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.

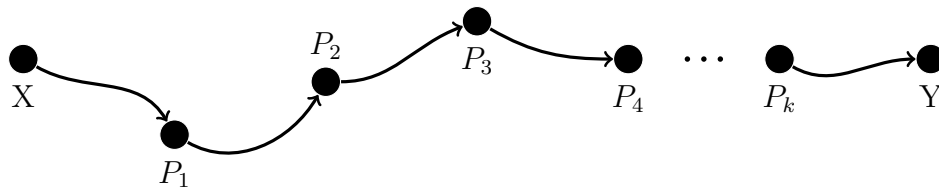


Figura 4 – Rota da cidade X para a cidade Y. Note que devemos passar necessariamente por cada parada para chegarmos ao destino. Ou seja, não podemos pular nenhum ponto.

Como ilustrado na Figura 4, 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 5 – 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 character;
3     struct celula *prox;
4 };
5
6 typedef struct celula 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 int inserir (Celula **c, char elem) {
2     Celula *nw = malloc(sizeof(Celula));
3     if(nw == NULL) return 0;
4
5     nw->character = elem;
6     nw->prox = *c;
7     *c = nw;
8
9     return 1;
10 }
```

Note que a lista a lista retorna um valor do tipo inteiro. Esse retorno é para indicar se a criação a inserção foi realizada com sucesso. Como em C não existe tipo *boolean*, consideramos *TRUE* com 1 e *FALSE* com 0.

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:

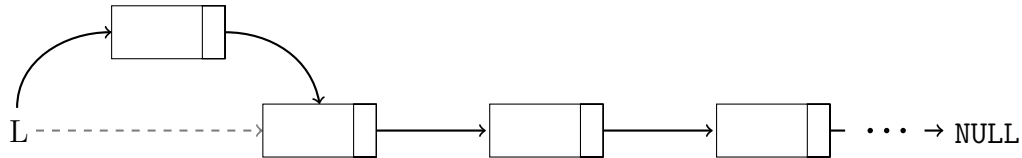


Figura 6 – Inserção de um elemento.

Exercício 5. *Implemente a inserção no final da lista.*

Exercício 6. *Implemente a inserção na k -ésima posição da lista.*

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

Exercício 7. *Implemente, ainda que não seja eficiente, a função de imprimir em ordem inverso.*

Exercício 8. *Implemente uma função que imprime a partir do elemento na posição x até o da posição y .*

3.1.4 Função de busca

A busca segue o mesmo princípio 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 Celula* buscar (Celula *c, char valor) {
2     Celula *p = c;
3     while (p != NULL) {
4         if(p->caracter == valor)
5             return p;
6         p = p->prox;
7     }
8     free(p);
9     return NULL;
10 }
```

Exercício 9. *Implemente a versão recursiva da busca.*

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:

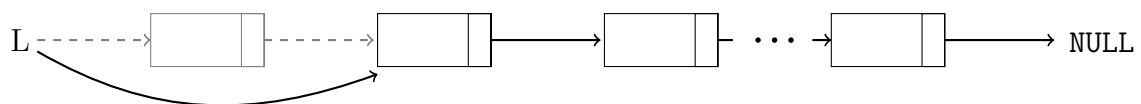


Figura 7 – Remoção do primeiro elemento.

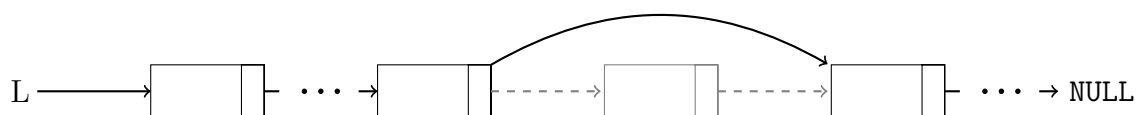


Figura 8 – Remoção do k-ésimo elemento.

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 está na lista e retornamos a lista original. Caso encontre um elemento, aplicamos o caso que a satisfaz, como descrito anteriormente.

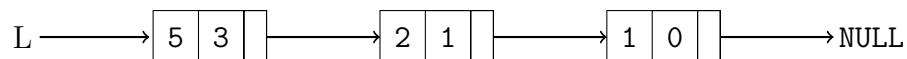

```

1  int remover(Celula **c, char valor)
2  {
3      Celula *anterior = NULL;
4      Celula *p = *c;
5      while (p != NULL && p->caracter != valor) {
6          anterior = p;
7          p = p->prox;
8      }
9
10     if (p == NULL)
11         return 0;
12
13
14     if (anterior == NULL)
15         *c = p->prox;
16     else
17         anterior->prox = p->prox;
18
19     free(p);
20     free(anterior);
21
22     return 1;
23 }

```

Exercício 10. *Implemente a versão recursiva da remoção.*

Exercício 11. *Podemos representar um polinômio de qualquer grau em uma estrutura de lista encadeada, onde cada célula contém o coeficiente, o expoente e a referência para o próximo termo. Por exemplo, o polinômio $5x^3 + 2x + 1$ teria a seguinte representação:*



- (a) *Faça uma função para criar um polinômio, inserindo os elementos em ordem decrescente em relação ao expoente do polinômio (como no exemplo, a última célula será a de expoente 0 e a primeira será a de maior expoente).*
- (b) *Faça uma função que receba dois polinômios e crie um terceiro resultante da soma deles.*

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.

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

3.3 Listas circulares

As listas circulares possuem uma estrutura semelhante ao que foi descrito até então, porém, existe um adicional: o último elemento passa a apontar para o primeiro, criando assim um ciclo.

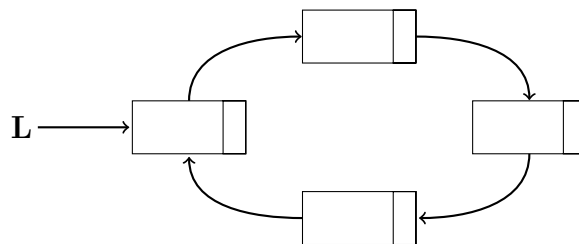


Figura 9 – Representação de uma lista circular.

Para implementarmos as operações em listas circulares partimos das operações já implementadas, realizando modificações.

A *struct* da lista circular é a mesma das listas encadeadas, uma vez que ainda precisamos armazenar o dado e o endereço do próximo nó. Da mesma forma é a inicialização, pois uma lista vazia deve apontar para o *NULL*.

Na inserção, o sucessor do último elemento da lista é o nó recém adicionado (no ciclo, o último nó é o mais distante do endereço da lista). Para inserir, devemos observar dois casos: se for a primeira inserção, ele deve apontar para si mesmo (isso evita de usarmos o *NULL* e nos ajudará nas próximas inserções); e para os outros casos, o sucessor do novo elemento deve apontar para o sucessor do primeiro nó, e, com isso, o primeiro nó passa a ser o último.

Assim como nas listas encadeadas, a inserção vai retornar um inteiro que indique se a inserção foi feita corretamente.

```

1 int inserir (Celula **c, char elem) {
2     Celula *nw = malloc(sizeof(Celula));
3     if(nw == NULL)
4         return 0;
5     nw->caracter = elem;
6
7     if(*c == NULL) {
8         nw->prox = nw;
9     } else {
10        nw->prox = (*c)->prox;
11        (*c)->prox = nw;
12    }
13
14    *c = nw;
15
16    return 1;
17 }

```

Para percorrer a lista, devemos adaptar a condição de parada do laço. Em vez de ser *NULL* o final do percurso, usaremos o primeiro elemento. Quando o laço atingi-lo, já percorremos toda a lista.

```

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

```

Dessa forma, apresentamos a lista da forma em que a função de inserção funciona. Ou seja, se fazemos:

```

1 // ...
2 Lista *l = inicializar();
3 inserir('1', l);
4 inserir('8', l);
5 inserir('9', l);
6
7 imprimir(l);
8 // ...

```

O resultado produzido é 918. Caso seja necessário imprimir na ordem que adicionamos os elementos, devemos implementar a função **imprimir** da seguinte forma:

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

Exercício 12. *Implemente a busca e remoção de listas circulares.*

3.4 Listas duplamente encadeadas

A lista duplamente encadeada é uma versão mais completa e complicada da lista encadeada simples. Antes, lidávamos apenas com o sucessor de um elemento, dessa forma, estávamos limitado por uma direção. Nessa nova versão, temos dois ponteiros no nosso nó: um para o próximo elemento e outro para o anterior. A vantagem do seu uso é a liberdade ao percorrer entre os elementos da lista. Uma vez que, na lista encadeada simples, não conseguimos, de forma eficiente, percorrer os elementos em ordem inversa^[4]. Com o acesso ao antecessor, facilitamos também a remoção de elemento, pois não seria preciso mais guardar o elemento anterior na busca.

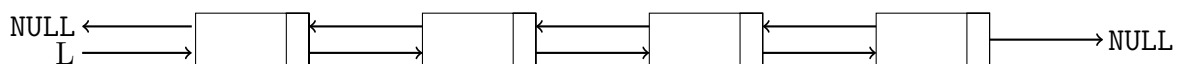


Figura 10 – Representação de uma lista duplamente encadeada.

Na estrutura da lista duplamente encadeada adicionamos mais um ponteiro.

```
1 struct celula {
2     char caracter;
3     struct celula *prox;
4     struct celula *ant;
5 };
6
7 typedef struct celula Celula;
```

Para inserirmos um valor, devemos adaptar a lista encadeada simples, adicionando o ponteiro do nó anterior. Como a inserção é feita no início da lista, o endereço anterior é sempre *NULL* (se ele é o primeiro, não existe um antecessor). Além disso, devemos verificar se a lista é vazia.

```

1 int inserir (Celula **c, char elem) {
2     Celula *nw = malloc(sizeof(Celula));
3     if(nw == NULL)
4         return 0;
5
6     nw->caracter = elem;
7     nw->prox = *c;
8     nw->prox = NULL;
9
10    if(*c != NULL)
11        (*c)->ant = nw;
12
13    *c = nw;
14
15    return 1;
16 }

```

Para percorrer a lista basta usarmos a mesma implementação da lista encadeada. Mas, se quisermos percorrer na ordem inversa, precisamos ir ao final da lista e usar o ponteiro *ant* para voltar:

```

1 void imprimirR(Celula *c) {
2     Celula *p = c;
3
4     while(p->prox != NULL)
5         p = p->prox;
6
7     while (p != NULL) {
8         printf("%c\n", p->caracter);
9         p = p->ant;
10    }
11    free(p);
12 }

```

Exercício 13. *Implemente a função de busca e de remoção da lista duplamente encadeada.*

Exercício 14. *Implemente as operações básicas (inserção, remoção e busca) da lista circular duplamente encadeada (a junção das duas últimas listas estudadas).*

3.5 DESAFIO: Implementação da BIGINT

Uma boa forma de exercitar um tópico de estudo é aplicá-lo como solução para um outro problema. Dessa forma, temos uma visão diferente daquilo que foi aprendido.

A estrutura **BIGINT** é uma representação de números a partir de caracteres em uma lista, onde os caracteres permitidos são os dígitos '0' à '9' e '-', que vem antes dos números negativos. Dessa forma, um número é a concatenação de caracteres organizados em uma lista. Por exemplo, para 1589 temos a lista:

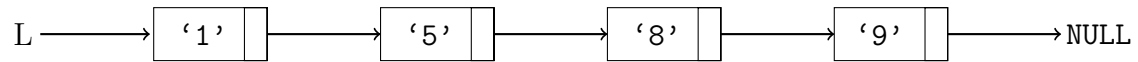


Figura 11 – Representação dos números no BIGINT em uma lista encadeada simples.

A partir disso, podemos implementar várias operações básicas, como soma e multiplicação. Mas, como implementar tais operações lidando com uma lista de caracteres? Para isso, devemos aplicar as operações aos dígitos individualmente, convertendo-o para inteiro.

Por exemplo, para realizarmos a soma, precisamos converter o último dígito das duas listas para inteiro, somá-los e converter o resultado para *char*, e isso para todo os algarismos. Vale salientar que devemos considerar todos os casos de uma soma de números. Por exemplo, quando um número possuir mais dígito do que o outro ou quando devemos levar a soma para o próximo algarismo.

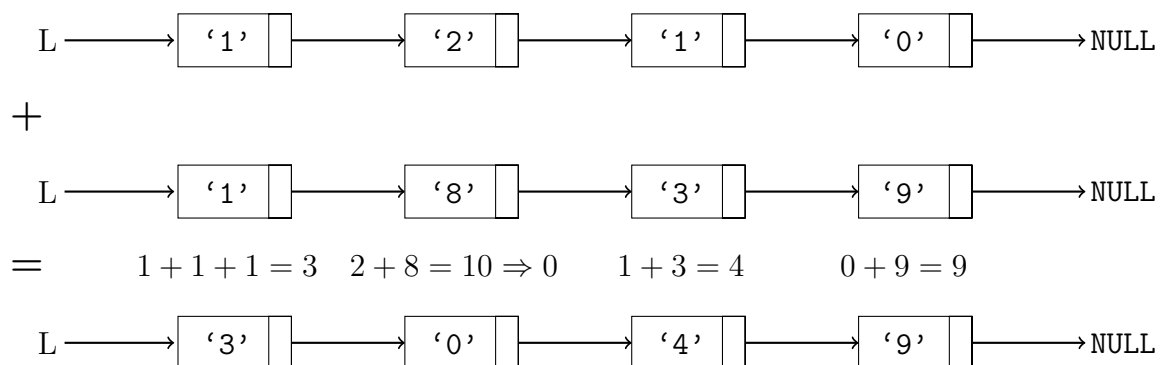


Figura 12 – Representação da soma de dois números que representam um número.

Dado essa breve explicação do problema, vamos construir as funções das operações e da estrutura. Porém, essa parte será o nosso desafio. Implemente as operações de soma, multiplicação e exponenciação. Utilize a estrutura de lista que desejar.

Com essa atividade, temos um exemplo de como utilizar as listas em um problema que vai além das operações básicas de listas.

4 Pilha

As pilhas são estrutura de dados onde o último elemento a ser inserido será o primeiro a ser removido. Essas estruturas são denominadas LIFO (*last-in, first-out*). Note que essa estrutura coloca uma restrição em suas operações. Na pilha, só temos

acesso ao último elemento adicionado e, dessa forma, para acessarmos outros elementos, devemos retirar os adicionados após ele. Por exemplo, em uma pilha com 5 elementos, para acessarmos o terceiro, devemos retirar o quinto e depois o quarto.

Podemos exemplificar essa estrutura com uma pilha de livros. A adição de livros é sempre feita no topo, colocando um em cima do outro. A remoção é da mesma forma. Para acessar um livro que esteja no meio da pilha, removemos todos os acima dele. Se tentarmos adicionar, remover ou acessar livros que não sejam no topo, provavelmente, eles irão cair. Logo, se quisermos alcançar um livro na pilha que não esteja no topo, deve-se remover todos os livros superiores a ele, independente do objetivo.

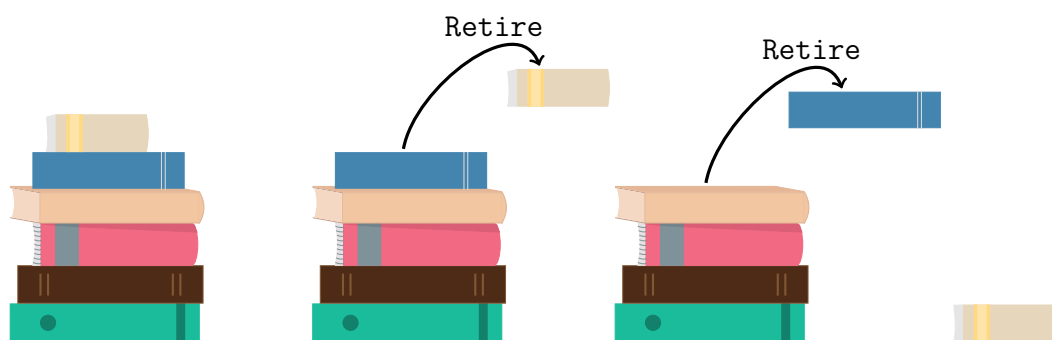


Figura 13 – Para alcançarmos o quarto livro dessa pilha, devemos retirar o sexto e quinto. Só depois disso, podemos fazer o que quisermos com o livro. Fonte: Autor.

Semelhantemente, a estrutura de pilha realiza operações apenas no topo. Por causa disso, essa estrutura é bastante simples de implementar e é muito utilizada em diversas aplicações, como por exemplo, no recurso de avançar/voltar em páginas de navegadores web e o refazer/desfazer de alguns editores.

4.1 Implementação da pilha e suas operações

Com o conceito definido, podemos partir para a implementação de algumas operações. Iremos implementar a nossa pilha a partir de um vetor e de uma lista encadeada. Usualmente a inserção é denominada *push* e a remoção *pop*.

4.1.1 Implementação com um array

O uso de array é extremamente simples e sua construção é quase que imediata. O que precisamos na estrutura é de um array, o topo do array e o tamanho máximo de elementos

```
1 struct pilha {  
2     int total;  
3     int topo;
```

```

4     char *character;
5 };
6
7 typedef struct pilha Pilha;

```

Na inicialização, além da pilha, devemos alocar memória para o array a partir do argumento passado. Além disso o topo do array, para representar que a pilha está vazia, é -1.

```

1 Pilha* inicializar (int tot) {
2     Pilha* aux = malloc(sizeof(Pilha));
3     aux->character = malloc(sizeof(char)*tot);
4
5     aux->topo = -1;
6     aux->total = tot;
7
8     return aux;
9 }

```

A inserção de elemento é dado em um espaço livre, enquanto houver, no array. Devemos verificar se o array está cheio. Se estiver cheio, retornamos 0 indicando que não foi inserido nenhum elemento, caso não esteja, inserimos e retornamos o valor 1.

```

1 int push (Pilha *p, char valor) {
2     if(p->topo == p->total - 1) {
3         return 0; // Overflow
4     }
5
6     p->topo++;
7     p->character[p->topo] = valor;
8     return 1;
9 }

```

A remoção é feita no último elemento adicionado do array. Basta decrementar a variável topo, desconsiderando o elemento.

```

1 char pop (Pilha *p) {
2     if(p->topo == -1) {
3         return -1; // UnderFlow
4     }
5     return p->character[p->topo--];
6 }

```

Como retornamos o caracter removido, para percorrer a pilha basta realizarmos um série de remoções.

E é simplesmente isso! Se quisermos tornar nossa implementação ainda mais simples, as operações de *pop* e *push* podem ser desenvolvidas no escopo da função principal, sem utilizar estruturas ou funções. A pilha será um array.

```
1 int main() {
2     char pilha[5]; // considere k=5, apenas para exemplificar
3     int t = 0;
4
5     pilha[t++] = 'a';
6     pilha[t++] = 'b';
7     pilha[t++] = 'c';
8     pilha[t++] = 'd';
9
10    t--;
11
12    return 0;
13 }
```

A variável t é o indicador do topo da pilha. Dessa forma, basta incrementar o t para adicionar elementos e decrementá-lo para remover.

Exercício 15. *Desenvolva a função **PULL**, que, basicamente, altera o elemento posicionado no topo da pilha.*

Exercício 16. *Desenvolva uma função para inverter a posição dos elementos de uma pilha.*

Exercício 17. *Faça uma função que receba uma pilha e retorne a quantidade de elementos.*

4.1.2 Implementação com um lista

O uso de uma lista encadeada para implementar uma pilha é dada quando não conhecemos o número de elementos que precisamos armazenar^[4]. A estrutura para os nós da pilha é semelhante ao das listas até então estudadas.

A inserção (*push*) na pilha é dada sempre no início da lista (da mesma forma que foi apresentado anteriormente, na seção 3.1.2). Dessa forma, o endereço da lista sempre aponta para o último elemento adicionado. A remoção do elemento é bastante simples: como removemos apenas o último adicionado, basta mudar o endereço da lista para o segundo nó.

A implementação das operações básicas é dada por:

```
1 struct pilha {
2     char caracter;
3     struct pilha *prox;
```

```

4  };
5
6  typedef struct pilha Pilha;
7
8  Pilha* inicializar() {
9      return NULL;
10 }
11
12 int push( Pilha **p, char valor) {
13     Pilha *nw = malloc(sizeof(Pilha));
14     if(nw == NULL)
15         return 0;
16
17     nw->caracter = valor;
18     nw->prox = *p;
19     *p = nw;
20
21     return 1;
22 }
23
24 char pop(Pilha** p) {
25     if(*p == NULL) {
26         return -1; // Underflow
27     }
28
29     Pilha *aux = *p;
30     *p = (*p)->prox;
31     char removido = aux->caracter;
32
33     free(aux);
34     return removido;
35 }

```

Exercício 18. *Implemente a função **PULL** para uma pilha em lista encadeadas.*

Exercício 19. *Desenvolva uma função que receba uma pilha e divida ela na metade em duas pilhas, sem modificar a ordem dos itens*

Exercício 20. *Desenvolva uma função que receba duas pilhas e coloque a segunda no topo da primeira, sem alterar a ordem dos elementos e deixando a segunda pilha vazia.*

Exercício 21. *Desenvolva uma função que receba duas pilhas e junte os elementos em uma única lista, intercalando os elementos.*

Observe que essas implementações não são as únicas. Por exemplo, na função *pop*, o ideal seria retornar o valor retirado do topo. Dessa forma, não precisamos consultar o topo antes de deletar o elemento, ao percorrer a pilha.

4.2 DESAFIO: Balanceamento de Parênteses

Uma aplicação das pilhas é utilizá-la para o balanceamento de parênteses, ou seja, para verificar se, em uma dada string, para cada abertura de parêntese existe um fechamento.

Por exemplo, a string `(a(-),a1)` está com parênteses balanceados, mas `(akd,))` está desbalanceado.

Com o uso de pilha, esse problema é facilmente resolvido. Quando achamos uma abertura, empilhamos; ao achar o fechamento correspondente, desempilhamos. Se repetirmos esse procedimento até o final da entrada, e se a entrada estiver balanceada, a nossa pilha deverá estar vazia no final. E é só isso! Implemente essa operação e adicione esse mesmo balanceamento para colchetes e chaves.

5 Filas

As filas (queue) são estruturas de dados onde os primeiros a serem inseridos serão os primeiros a serem removidos. Essa estrutura é nomeada FIFO (*first in, first out*). Semelhante a pilha, as operações de inserção e remoção de elementos na fila possuem restrições. A adição de elemento é semelhante ao da pilha e é realizado no fim da estrutura e a remoção, é a partir do primeiro elemento adicionado, em contrapartida ao da pilha, que é no último.

Note que podemos fazer uma relação dessa estrutura de uma fila real. Imagine uma fila de um banco. Nessa situação, aqueles que chegarem primeiro devem ser logo atendidos. Seria bastante injusto se alguém que chegasse muito tempo depois dos outros fosse primeiramente atendido. Da mesma forma, em um *drive-thru* de um *fast-food*: aquele que chegar na frente será atendido primeiro que os outros.

Podemos, ainda, exemplificar o uso de filas a partir de uma rede de compartilhamento de impressora. Com essa rede, podemos realizar impressões em qualquer computador conectado a ela e, dessa forma, podemos reduzir custos. No entanto, existe um problema quando há mais de uma solicitação. Como podemos controlar as solicitações de usuários? E as "simultâneas"? Bem, o ideal é responder o primeiro que fez a solicitação e depois, o segundo e assim por diante. Mas para isso, é necessário um espaço na memória no computador que fique armazenados os arquivos enviados para impressora. Esse espaço na memória é denominado fila de impressão ou *spooler*.

A fila é uma estrutura bastante simples e utilizada na solução de diversos problemas práticos como, por exemplo, nos algoritmos de enfileiramento usados nos roteadores e, como ilustrado anteriormente, no controle de impressão.

Bem, dados os exemplos, vamos para as implementações.

5.1 Representação e implementação da fila

Como nas pilhas, vamos desenvolver duas versões de fila: um, com uso de vetores e outro, com listas encadeadas. Será implementado a operação de inserção (*enqueue*), e a remoção (*dequeue*). Será implementado uma fila que armazena caracteres.

5.1.1 Fila implementada em um array

Para implementarmos uma fila em vetores, precisaremos de três informações: o número total do array, a posição inicial da fila e a posição final. A posição inicial indica onde o elemento que pode ser removido está, e a posição final indica onde será inserido um elemento. A estrutura da fila, com uso de vetores, é definida como:

```
1 struct fila {
2     int total;
3     int inicio;
4     int fim;
5     char *character;
6 };
7
8 typedef struct fila Fila;
```

A inicialização é bastante simples: basta alocarmos memória para a *struct* fila e para o vetor, que armazena os caracteres, com o tamanho passado pelo usuário. Além disso, a variável *inicio* e *fim* serão iguais a zero.

```
1 Fila* inicializar (int tot) {
2     Fila* aux = malloc(sizeof(Fila));
3     aux->character = malloc(sizeof(char)*tot);
4
5     aux->inicio = aux->fim = 0;
6     aux->total = tot;
7
8     return aux;
9 }
```

Para exemplificar, suponha que construimos um vetor com oito posições na nossa fila. Vamos representá-la da seguinte forma:

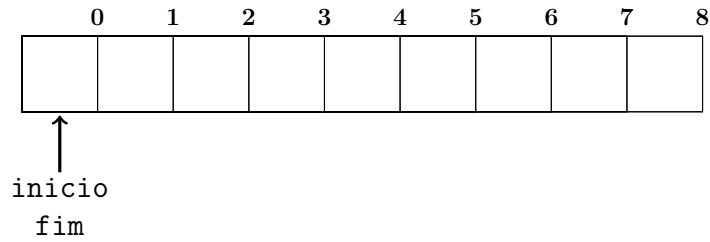


Figura 14 – Estrutura da fila em um vetor.

Observe que $inicio = fim = 0$. Isso indica que a fila está vazia e a fila inicia na posição 0. É importante salientar que o início nem sempre estará na posição 0. Isso é evidenciado na remoção.

Quando adicionamos elementos, a variável fim será incrementada, de forma que ocupa uma posição vazia onde será inserido um próximo elemento. Com isso, teremos que desperdiçar uma posição do array, mas dessa forma poderemos verificar se a fila está vazia ou cheia.

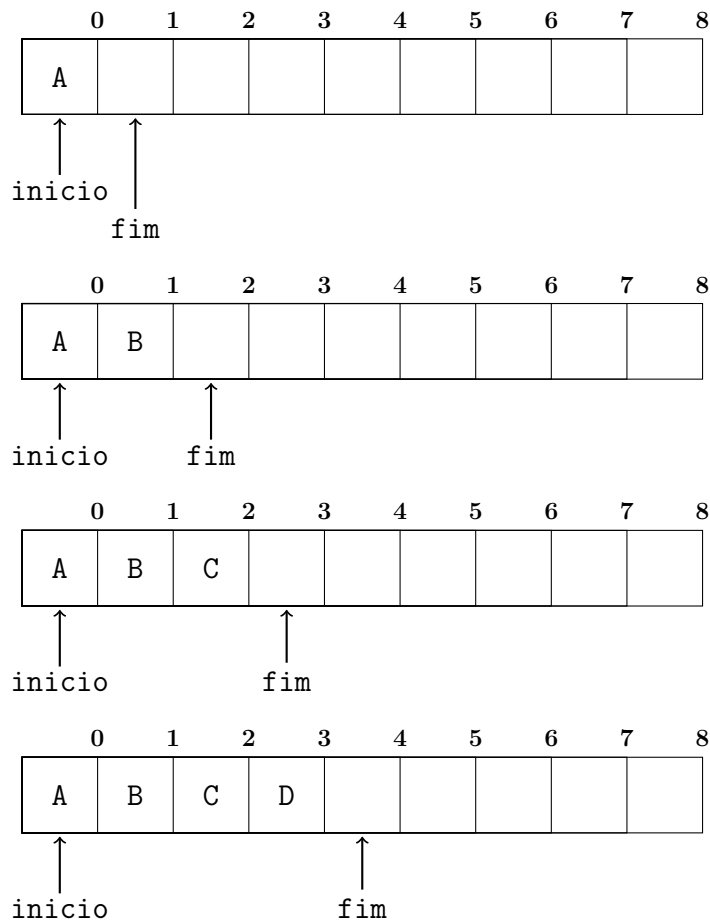


Figura 15 – Adição, em ordem, dos caracteres 'A', 'B', 'C' e 'D'.

Obviamente, com a adição de elementos, haverá um momento em que fim atingirá a última posição do array. Isso significa que a fila está cheia? Não necessariamente, pois

devemos nos atentar a remoção. Ao remover, *inicio* será incrementada e, dessa forma, os espaço anteriores ao *inicio* serão desconsiderados.

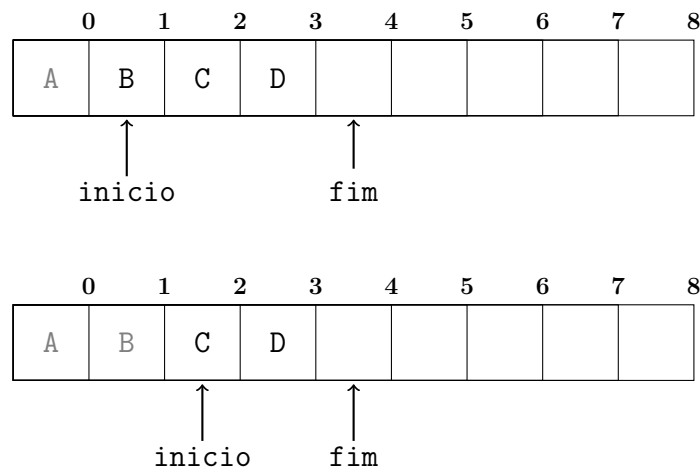


Figura 16 – Remoção dos caracteres ‘A’ e ‘B’.

Note que, no exemplo acima, existem duas posições no início do array que são desconsideradas. Para reaproveitarmos essas posições, poderíamos reorganizar nosso vetor, voltando cada posição, até ocupar os espaços livres. No entanto, isso seria extremamente custoso, uma vez que, em toda remoção, teríamos que realizar a realocação dos elementos.

Existe uma maneira mais simples de resolver isso. Basta utilizar um incremento com o operador módulo, de forma que o vetor seja circular. Isto é, ao chegarmos na última posição do array e incrementarmos, voltamos para o início do vetor. Para isso, em vez de incrementar com $(i + 1)$ usaremos $(i + 1) \% T$, onde T representa o final do nosso ciclo. Quando $i = T - 1$, temos $(T - 1 + 1) \% T = (T) \% T = 0$, voltando ao início.

Agora, vamos implementar a operação *enqueue*. Nela, vamos adicionar o elemento no *fim* e, depois disso, incrementá-lo. Além disso, devemos verificar se a fila já está cheia. Para isso, basta verificarmos se o *inicio* é sucessor do *fim*, uma vez que todas as posições estariam ocupadas (de *inicio* até chegar em *fim*, não tendo mais espaço entre eles).

```
1 int enqueue(Fila *f, char valor) {
2     int inc = (f->fim + 1) % f->total;
3     if(inc == f->inicio)
4         return 0;
5
6     f->caracter[f->fim] = valor;
7     f->fim = inc;
8     return 1;
9 }
```

Para a operação *dequeue*, devemos incrementar, usando o operador de módulo, o *inicio* e, dessa forma, desconsideramos o elemento mas a frente. Porém, antes disso,

precisamos verificar se a fila está vazia. Para isso, devemos certificar se *inicio* é igual a *fim* (como *fim* ocupa um espaço vazio, *inicio* também será vazio e, dessa forma, não temos elemento armazenado). Por fim, retornamos o caracter removido.

```
1 char dequeue(Fila *f) {
2     if(f->inicio == f->fim)
3         return -1; // Fila Vazia
4
5     char c = f->caracter[f->inicio];
6     f->inicio = (f->inicio + 1) % f->total;;
7     return c;
8 }
```

Caso seja necessário imprimir a fila, respeitando as restrições, devemos realizar várias chamadas de *dequeue*. Porém, para teste, podemos imprimir o vetor utilizando a seguinte função:

```
1 void imprimir(Fila *f) {
2     for (int i = f->inicio; i < f->fim; i = (i+1)%f->total) {
3         printf("fila[%i] = %c\n", i, f->caracter[i]);
4     }
5 }
```

Dessa forma, podemos verificar se nossas operações estão corretas, de forma que usamos o incremento com o operador módulo.

Exercício 22. *Desenvolva uma função que inverta a ordem dos elementos, respeitando as restrições da fila.*

Exercício 23. *conteúdo...*

5.1.2 Fila implementada em um lista

Agora, vamos implementar uma fila a partir de uma lista encadeada simples que armazene caracteres. Para isso, nossa estrutura deverá guardar o caracter e a posição do próximo nó. Além disso, temos que desenvolver uma estrutura adicional que armazena a posição inicial e final da fila. Essa estrutura facilitará no acesso dessas posições, para a inserção (fim) e remoção (inicio) de elementos. Poderíamos não implementar essa segunda estrutura, porém, na inserção, teríamos que percorrer toda a lista, o que seria custoso.

O estrutura do nó e da fila são definidos como:

```
1 struct celula {
2     char caracter;
3     struct celula *prox;
4 };
```

```

5
6 typedef struct celula Celula;
7
8 struct fila {
9     Celula *inicio;
10    Celula *fim;
11 }
12
13 typedef struct fila Fila;

```

Podemos representar essas estruturas da seguinte forma:

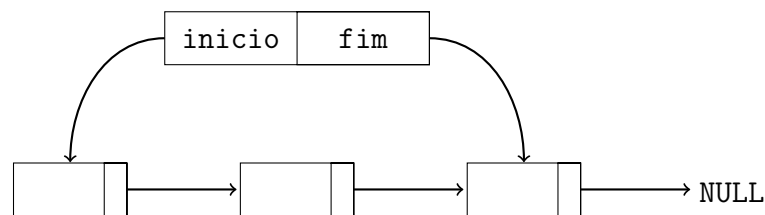


Figura 17 – Representação da lista celula e da estrutura fila.

Para inicializar a estrutura, devemos alocar memória para a fila e definir os ponteiros *inicio* e *fim* como nulo, pois a fila está vazia.

```

1 Fila *inicializar() {
2     Fila *f = malloc(sizeof(Fila));
3
4     f->inicio = f->fim = NULL;
5     return f;
6 }

```

A inserção de elementos (enqueue) é dada com a adição do elemento no final da lista. Como temos o endereço do último elemento, essa operação fica mais simples.

```

1 int enqueue(Fila *f, char valor) {
2     Celula *aux = malloc(sizeof(Celula));
3     if(aux == NULL)
4         return 0;
5
6     aux->character = valor;
7     aux->prox = NULL;
8
9     if(f->fim != NULL)
10        f->fim->prox = aux;
11
12    f->fim = aux;

```



```

13
14     if (f->inicio == NULL)
15         f->inicio = f->fim;
16     return 1;
17 }

```

Nessa operação, devemos criar uma nova célula que armazena o valor que desejamos inserir. O endereço do próximo desse novo nó deve ser `NULL`, uma vez que representa o último elemento adicionado. Verificamos se já existe um elemento na fila e, caso exista, o último elemento anterior terá o *prox* o novo nó adicionado. Dessa forma, o novo *fim* será o nó criado. Caso a fila esteja vazia, e adicionado o elemento (sendo este o primeiro), o *inicio* deve apontar também para este elemento (ele será o início e fim).

A operação dequeue é uma versão simplificada da remoção das listas encadeadas. Nela, não precisamos percorrer a lista em busca do elemento adicionado. Basta remover o primeiro nó e mudar o endereço do *inicio* para o segundo. Mas precisamos ainda realizar algumas verificações.

```

1 char dequeue(Fila *f) {
2     if (f->inicio == NULL)
3         return -1;
4
5     char v = f->inicio->caracter;
6
7     Celula *lixo = f->inicio;
8
9     f->inicio = f->inicio->prox;
10    if (f->inicio == NULL)
11        f->fim = NULL;
12
13    free(lixo);
14    return v;
15 }

```

Inicialmente, devemos verificar se a lista está vazia, caso *inicio* não aponte para nenhum elemento, não foi adicionado elemento (temos o mesmo efeito para o *fim*). Além disso, verificamos se, após a remoção do nó, o *inicio* aponta para `NULL` e, caso aponte, definimos *fim* também como nulo, de forma que a fila seja vazia. Retornando o caracter removido podemos percorrer a fila realizando remoções consecutivas.

Exercício 24. Respeitando as restrições das filas, faça uma função que receba uma fila e retorne a quantidade de itens da fila.

Exercício 25. *Desenvolva uma função que receba duas filas e coloque a segunda no final da primeira fila, sem alterar a ordem dos elementos e deixando a segunda pilha vazia.*

Exercício 26. *Desenvolva uma função que receba uma fila e divida ela na metade em duas filas, sem modificar a ordem dos itens.*

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>>.