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 bastante dificuldade em compreender, em um primeiro momento. 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:

Algoritmo 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:

Algoritmo 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:

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 ← 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.

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

$\binom{0}{0}$	1
$\binom{1}{0}\binom{1}{1}$	1 1
$\binom{2}{0}\binom{2}{1}\binom{2}{2}$	1 2 1
$\binom{3}{0}\binom{3}{1}\binom{3}{2}\binom{3}{3}$	1 3 3 1
$\binom{4}{0}\binom{4}{1}\binom{4}{2}\binom{4}{3}\binom{4}{4}$	1 4 6 4 1
...	...

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,*

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

.

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 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:

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 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 contrapartida, 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.

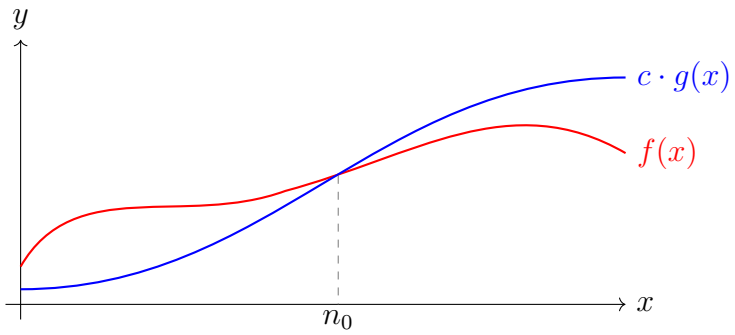


Figura 2 – No ponto n_0, $c \cdot g(x)$ domina $f(x)$, dessa forma $f(x) \in O(g(x))$.

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.

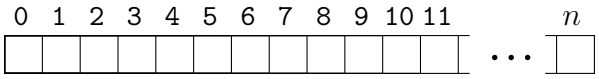


Figura 3 – Estrutura do vetor v na memória

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.

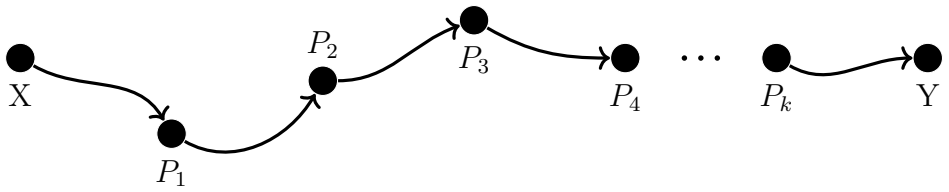


Figura 4 – Rota da cidade X para a cidade Y. Note que devemos passar necessariamente por cada parada para chegarmos ao destino. Ou seja, não podemos pular nenhum ponto.

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**.

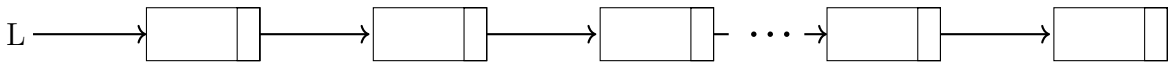


Figura 5 – Ilustração de uma lista encadeada. O endereço da lista é representada por um ponteiro que aponta para o primeiro elemento.

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 caracter;
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->caracter = 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:

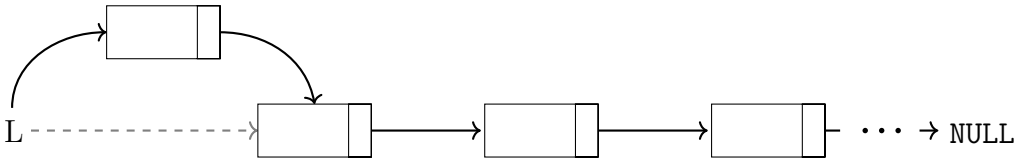


Figura 6 – Inserção de um elemento.

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 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  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:

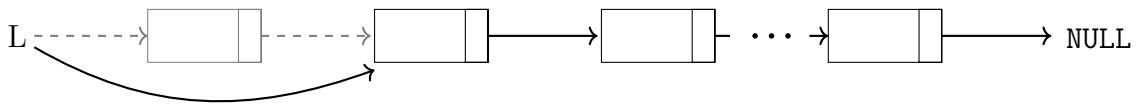


Figura 7 – Remoção do primeiro elemento.

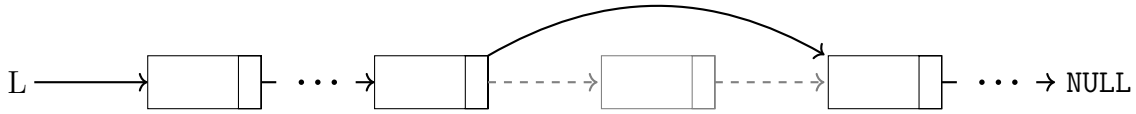


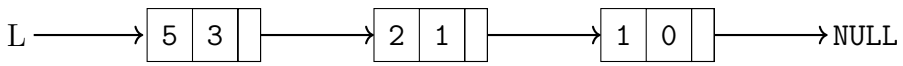
Figura 8 – Remoção do k-ésimo elemento.

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

```c
int remover(Celula **c, char valor)
{
   Celula *anterior = NULL;
   Celula *p = *c;
   while (p != NULL && p->caracter != valor) {
      anterior = p;
      p = p->prox;
   }

   if (p == NULL)
      return 0;


   if (anterior == NULL)
      *c = p->prox;
   else
      anterior->prox = p->prox;

   free(p);
   free(anterior);

   return 1;
}
```

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 o de expoente 0 e a primeira será o 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.

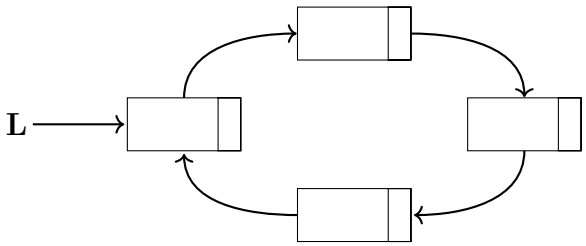


Figura 9 – Representação de uma lista circular.

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.

```c
int inserir (Celula **c, char elem) {
   Celula *nw = malloc(sizeof(Celula));
   if(nw == NULL)
      return 0;
   nw->caracter = elem;

   if(*c == NULL) {
      nw->prox = nw;
   } else {
      nw->prox = (*c)->prox;
      (*c)->prox = nw;
   }

   *c = nw;

   return 1;
}
```

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 todo a lista.

```c
void imprimir(Celula *c) {
   Celula *p = c;

   do {
      printf("%c\n", p->caracter);
      p = p->prox;
   } while (p != c);
}
```

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

```c
   //...
   Lista *l = inicializar();
   inserir('1', l);
   inserir('8', l);
   inserir('9', l);

   imprimir(l);
   // ...
```

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, lidavamos 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.

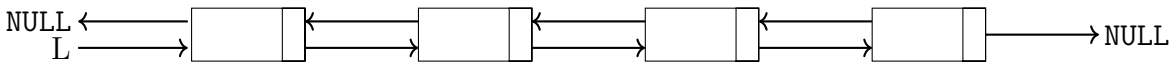


Figura 10 – Representação de uma lista duplamente encadeada.

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 ínicio 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.

```c
int inserir (Celula **c, char elem) {
   Celula *nw = malloc(sizeof(Celula));
   if(nw == NULL)
      return 0;

   nw->caracter = elem;
   nw->prox = *c;
   nw->prox = NULL;

   if(*c != NULL)
      (*c)->ant = nw;

   *c = nw;

   return 1;
}
```

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:

```c
void imprimirR(Celula *c) {
   Celula *p = c;

   while(p->prox != NULL)
      p = p->prox;

   while (p != NULL) {
      printf("%c\n", p->caracter);
      p = p->ant;
   }
   free(p);
}
```

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:

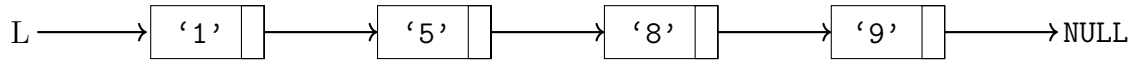


Figura 11 – Representação dos números no BIGINT em uma lista encadeada simples.

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.

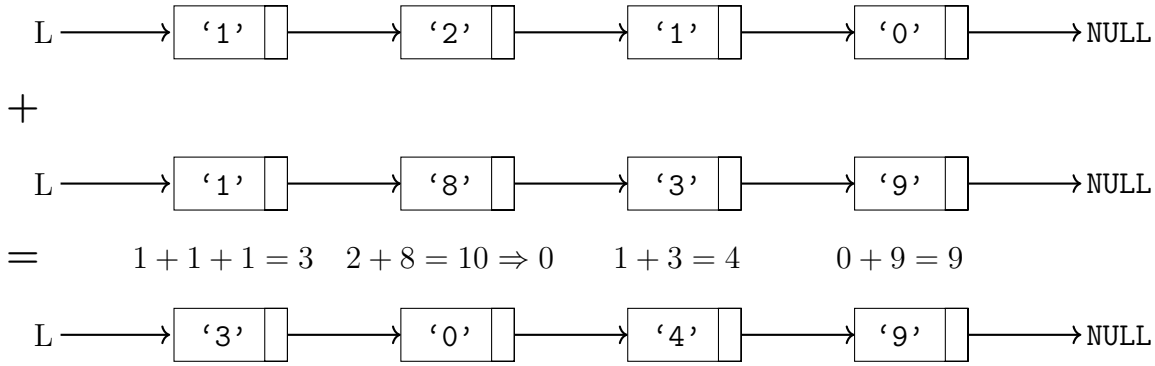


Figura 12 – Representação da soma de dois números que representam um número.

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 exponeciação. 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.

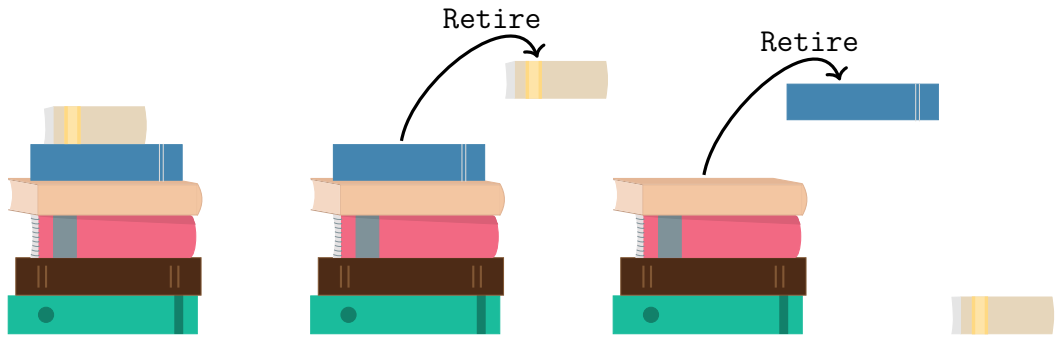


Figura 13 – Para alcançarmos o quarto livro dessa pilha, devemos retirar o sexto e quinto. Só depois disso, podemos fazer o que quisermos com o livro. Fonte: Autor.

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 *caracter;
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->caracter = 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->caracter[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->caracter[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 váriavel 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 posicão 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 ínicio 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(-),al)` estar 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 operações 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 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 *caracter;
6 };
7
8 typedef struct fila Fila;
```

A inicalização 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->caracter = malloc(sizeof(char)*tot);
4
5    aux->inicio = aux->fim = 0;
6    aux->total = tot;
7    return aux;
8 }
```

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

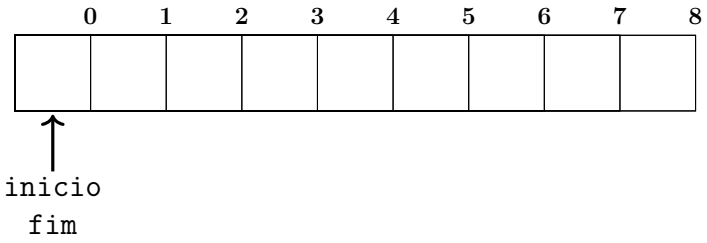


Figura 14 – Estrutura da fila em um vetor.

Observe que *inicio* = *fim* = 0. Isso indica que a fila está vazia e a fila inicia na posição 0. É importante salientar que o ínicio 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.

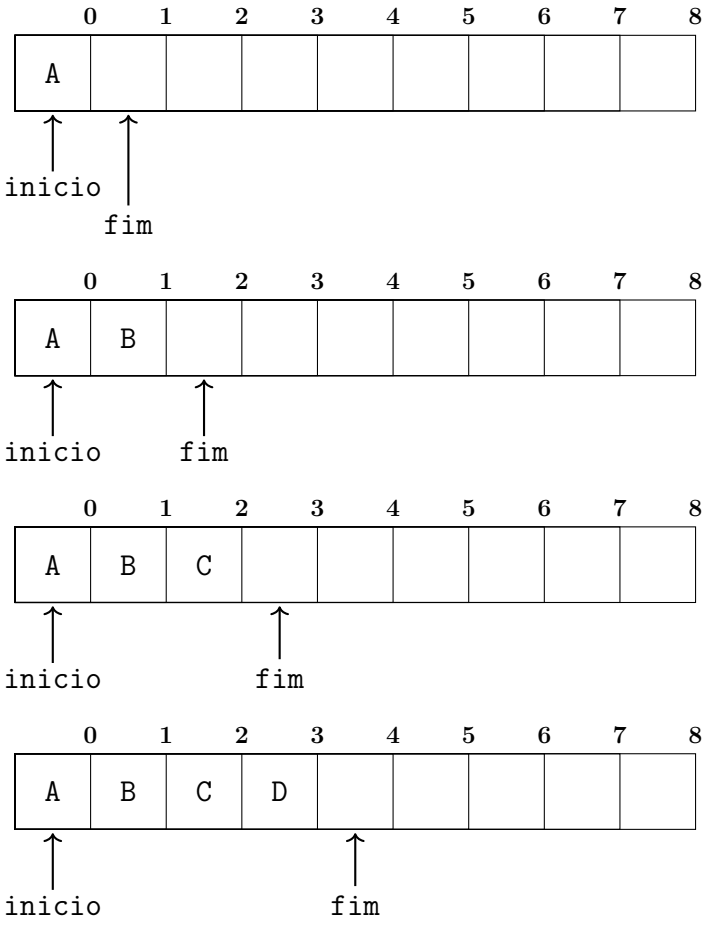


Figura 15 – Adição, em ordem, dos caracteres ‘A’, ‘B’, ‘C’ e ‘D’.

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.

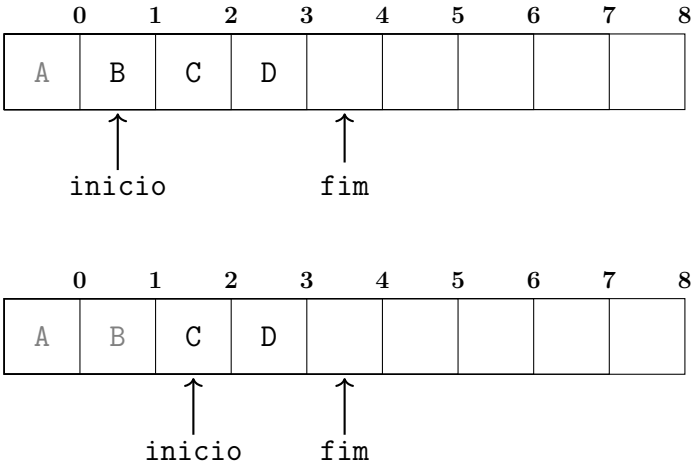


Figura 16 – Remoção dos caracteres 'A' e 'B'.

Note que, no exemplo acima, existem duas posições no ínicio 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 ínicio 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 ínicio.

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.

```
char dequeue(Fila *f) {
   if(f->inicio == f->fim)
      return -1; // Fila Vazia

   char c = f->caracter[f->inicio];
   f->inicio = (f->inicio + 1) % f->total;;
   return c;
}
```

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:

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

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.*

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 (ínicio) 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:

```
struct celula {
   char caracter;
```

```
3     struct celula *prox;
4  };
5  typedef struct celula Celula;
6
7  struct fila {
8     Celula *inicio;
9     Celula *fim;
10 }
11
12 typedef struct fila Fila;
```

Podemos representar essas estruturas da seguinte forma:

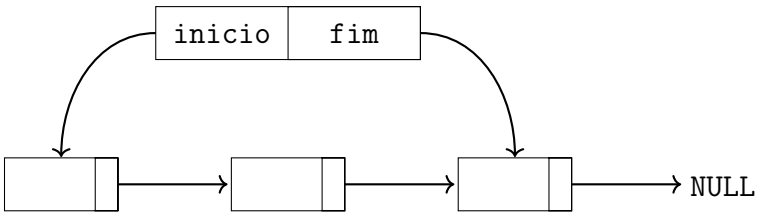


Figura 17 – Representação da lista celula e da estrutura fila.

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->caracter = 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 ínicio 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 23. *Respeitando as restrições das filas, faça uma função que receba uma fila e retorne a quantidade de itens da fila.*

Exercício 24. *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 25. *Desenvolva uma função que receba uma fila e divida ela na metade em duas filas, sem modificar a ordem dos itens.*

6 Árvores

Uma árvore é uma estrutura de dados que organiza seus elementos de maneira hierárquica, ou seja, existe um elemento "superior" e um "inferior"‘. Por exemplo, sua organização, de certa forma, é semelhante ao modo que uma empresa é estruturada: o chefe está no topo, logo abaixo estão os gerentes e, após esses, os seus subordinados.

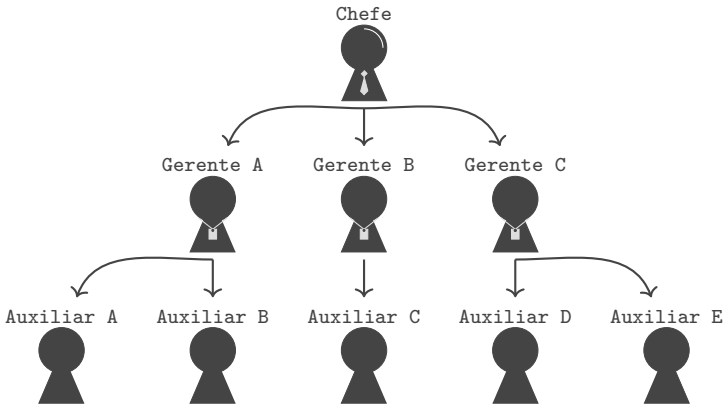


Figura 18 – Representação da hierarquia organizacional de uma empresa. Fonte: Autor.

Ao contrário das estruturas estudadas anteriormente, as árvores não são organizadas sequencialmente, ou seja, não possui um elemento "anterior" e um "sucessor". Geralmente, a terminologia utilizada vem das árvores genealógicas, onde utiliza-se os termos "pai" e "filho" para descrever os relacionamentos entre os nós[5].

6.1 Definição e representação de uma árvore

Uma árvore é um conjunto de diversos nós. Com exceção do elemento do topo, cada nó da árvore possui um ***pai*** e zero ou mais ***filhos***. O ***pai*** de um nó k, que não esteja no topo, é o elemento superior que se liga com uma subconjunto de elementos da árvore que o k está incluido. Os filhos de k, caso tenha, é o subconjunto em que o k faz ligação. Esse subconjunto é dita uma subárvore. O elemento do topo não possui um pai, sendo o ponto de partida da árvore e, devido isso, ele é dito a ***raiz***. O elemento que não possui ***filhos*** é nomeado como ***folha***.

Formalmente, definirmos uma árvore T como um conjunto de nós tal que[6]:

- $T = \emptyset$, ou seja, T não possui elementos, sendo dita vazia;

- ou existe o nó raiz, sendo o restante um conjunto vazio, ou ainda, $m \geq 0$ conjuntos disjuntos não vazios, as subárvores de r, cada qual sendo uma árvore.

Podemos representar graficamente uma árvore a partir da representação hierárquica, de forma que a raiz esteja no topo e as subárvores abaixo:

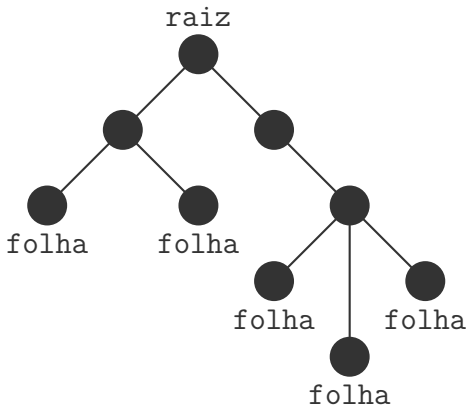


Figura 19 – Representação hierárquica de uma árvore.

Note que podemos relacionar sua estrutura com uma árvore invertida, ainda que, nesse exemplo, não tenha uma grande semelhança. Porém, é mais intuitivo relacionar a árvore com a definição de grafos. Este tema será elucidado futuramente.

Ademais, podemos representar uma árvore de outras formas. Com diagrama de inclusão, onde representamos os nós como círculos e seus filhos são incluidos internamente nesse círculo. Essa representação é a mais próxima da definição da árvore [6].

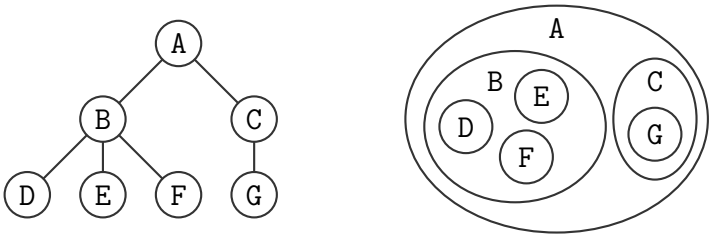


Figura 20 – À esquerda, representação hierárquica e, à direita, representação equivalente pelo o diagrama de inclusão

Uma outra representação, é por meio de aninhamento, onde representamoas a relação do pai e filho por meio de parênteses. Com isso, podemos representar a árvore da figura 20 da seguinte forma: (A(B(D, E, F)), C(G)). Esse modelo será útil na implementação do desafio (seção 6.7).

6.2 Propriedades de uma árvore

Devido sua forma de organizar os dados, as árvores possui diversas propriedades. Com isso, podemos inferir conceitos que são bastantes úteis na sua definição.

6.2.1 Grau dos nós da árvore e ordem

O grau de uma árvore representa a quantidade de ligações que um determinado nó possui, ou seja, o total de filhos desse nó.

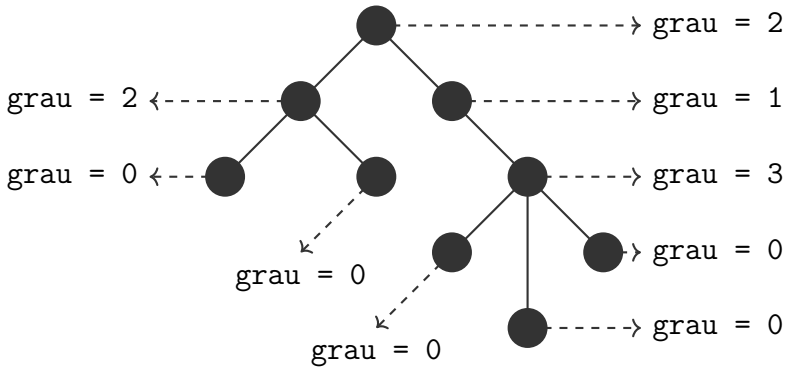


Figura 21 – Indicação do grau de todos os nós de uma árvore.

Podemos relacionar essa estrutura como uma árvore real, mas invertida. Observe que um nó de grau 0 é sempre uma folha.

Com o conceito de grau definido, podemos descrever o que é a ordem de uma árvore. A ordem é o número de maior grau de uma árvore. Na figura 21, a ordem é 3, uma vez que este é seu grau máximo.

6.2.2 Profundidade de um nó

A profundidade (ou altura) de um nó reflete a sua distância à raiz, contados os nós entre eles.

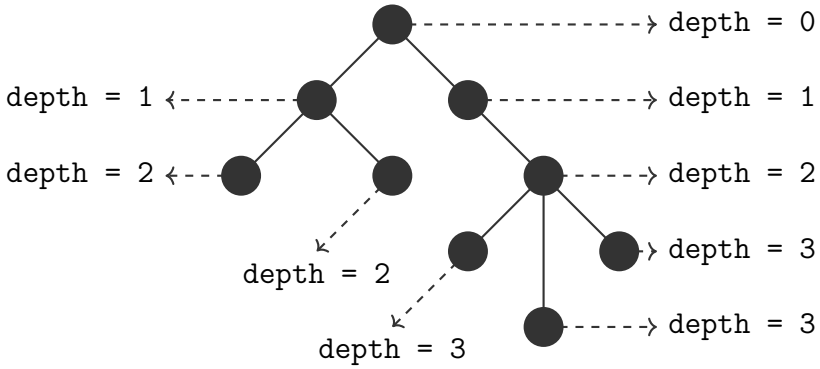


Figura 22 – Indicação da profundidade (*depth*) de todos os nós de uma árvore.

Note que a raiz terá profundidade 0. Os elementos sucessores à raiz possui profundidade 1, e os seu sucessores, profundidade 2 e assim por diante.

6.2.3 Nível e altura de uma árvore

O nível de uma árvore é o conjunto de nós que possui a mesma profundidade. Os nós desse conjunto são ditos irmãos.

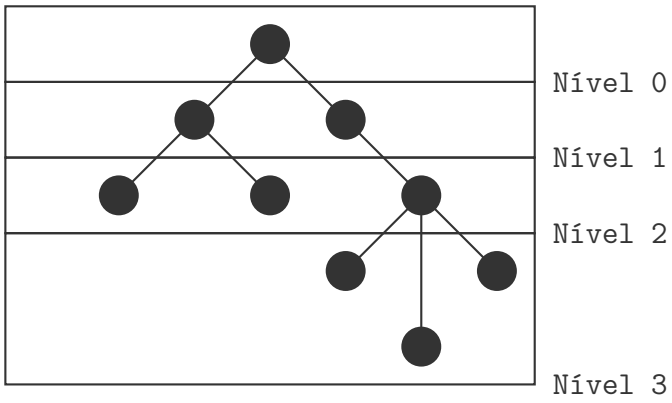


Figura 23 – Representação dos níveis

A altura de uma árvore é o valor do seu último nível.

6.2.4 Árvore cheia e completa

Uma árvore é dita cheia se, quando um nó possui alguma subárvore vazia, então ele está no último nível. A versão mais frouxa dessa definição, a árvore é completa quando possui todos os seus níveis cheios, exceto o último. Observe que toda árvore cheia é completa, mas nem toda árvore completa é cheia.

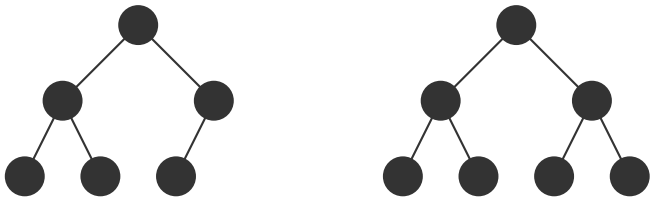


Figura 24 – À esquerda temos uma árvore completa e, à direita, uma árvore cheia.

6.2.5 Árvore balanceada

Uma árvore é dita balanceada quando mantém a profundidade de todos os nós em $O(logn)$, ou seja, os nós devem estar bem distribuidos. Existe outras condições de equilibrio dependendo da estrutura implementada (veja a seção 6.6 e compare árvore AVL e árvore rubro-negra, ambas balanceadas).

6.3 Árvore binária

Até então definimos árvore com grau indefinido, ou seja, com zero ou mais filhos. Porém pode ser necessário limitar a ordem da árvore, dependendo da aplicação. As árvores de ordem 2 é a mais conhecida e utilizada. Essa árvore denominamos árvore binária.

Note que, até então, apenas descrevemos conceitos teóricos de uma árvore. Isso significa que não estávamos preocupado com a implementação em si. Doravante, iremos implementar a nossa estrutura de dados e suas operações. Mas antes, é importante destacar alguns conceitos. Definimos:

> "Uma árvore binária é um conjunto finito de elementos que está vazio ou é particionado em três subconjuntos disjuntos. O primeiro subconjunto contém um único elemento, chamado raiz da árvore. Os outros dois subcon- juntos são em si mesmos árvores binárias, chamadas subárvores esquerda e direita da árvore original. Uma subárvore esquerda ou direita pode estar vazia."Tenenbaum, Langsam e Augenstein, 2004[7].

Em outras palavras, uma árvore binária limita a quantidade de filhos máxima de cada nós em dois.

Agora, vamos apresentar a representação de uma árvore binária. Para isso, podemos utilizar os mesmos modelos apresentados anteriormente (seção 6.1). No entanto, vamos nos limitar a representação hierárquica, devido a sua simplicidade e por ser comumente utilizada. Na figura a seguir, temos um exemplo de árvore binária e outra não binária.

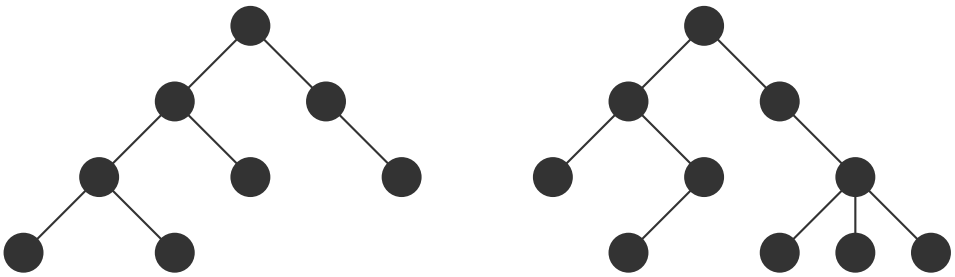


Figura 25 – Exemplo de uma árvore binária (esquerda) e de uma árvore não binária (direita).

6.3.1 Aplicações de árvore binárias

Podemos utilizar as árvores binárias em diversas aplicações. Por exemplo, é utilizado na representação de expressões aritmética contendo operandos e operadores binários (isso vale para árvores de qualquer ordem e, com isso, podemos representar qualquer expressão). Por exemplo, considere a expressão $(5 + 12) \div 15$. Por meio de uma árvore binária podemos representar da seguinte forma:

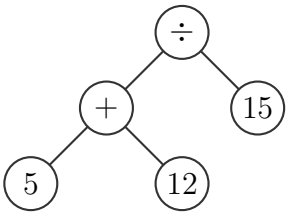


Figura 26 – Representação da expressão $(5 + 12) \div 15$.

Note que o operador com maior precedência é a mais interno na árvore. Isso é útil ao executarmos uma travessia completa (seção 6.4) na árvore, para calcular a expressão. O uso de árvores para esse fim é mais evidente no processo de análise de um compilador, na construção de uma árvore sintática.

Ademais, a árvore binária é utilizada em *heaps* binária, nas árvores de codificação de Huffman, na árvore BSP, etc. E ainda, podemos utilizá-la para construir uma estrutura famosa e bastante utilizada: a árvore binária de busca.

6.3.2 Árvore binária de busca

A árvore binária de busca (ou BST, *binary search tree*) é uma estrutura que estabelece propriedades na organização de seus nós. Para qualquer nó n, a subárvore esquerda armazena os elementos menores que n, enquanto que a subárvore direita armazena os elemenos maiores a n. Note que o valor (ou a chave) armazenado é comparável e, dessa forma, temos uma ordenação de elementos.

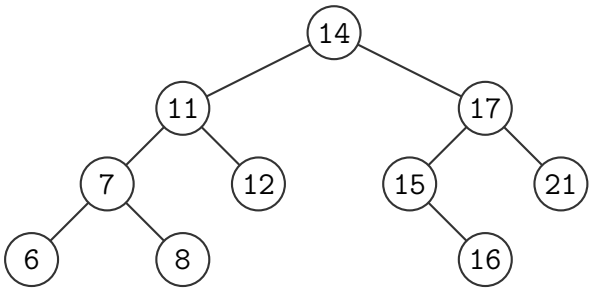


Figura 27 – Representação de uma árvore binária de busca.

6.3.3 Implementação da BST

Agora, vamos implementar as operações de inserção, remoção e busca de elementos em uma BST. A estrutura da BST é dado por:

```
1 struct no {
2     int valor;
3     struct *no dir;
4     struct *no esq;
```

```
};

typedef struct no No;
```

Na estrutura, armazenamos o dado propriamente dito, além da posição do filho esquerdo, onde armazenamos os elementos menores que o nó, e do filho direito, que armazena os elementos maiores. Como exemplo, iremos construir um BST que armazene inteiros.

6.3.3.1 Inicialização

Para inicializarmos a estrutura basta retornar *NULL*.

```
No* inicializar() {
   return NULL;
}
```

6.3.3.2 Busca

Essa operação aproveita bem da definição da BST. Utilizaremos a recursão ao nosso favor. A busca inicia-se na raiz e verifica se o valor buscado é o mesmo da raiz. Caso não seja, verifica se é maior ou menor. Caso o valor seja menor, segue pela subárvore esquerda, no contrário, pela subárvore direita. Esse procedimento é repetido recursivamente até encontrar um nó com o valor buscado ou chegar em uma folha.

```
No* buscar(No *arv, int v) {
   if (arv == NULL) return NULL;
   else if (arv->valor == v) return arv;
   else if (arv->valor > v) return busca (r->esq, v);
   else return busca (r->dir, v);
}
```

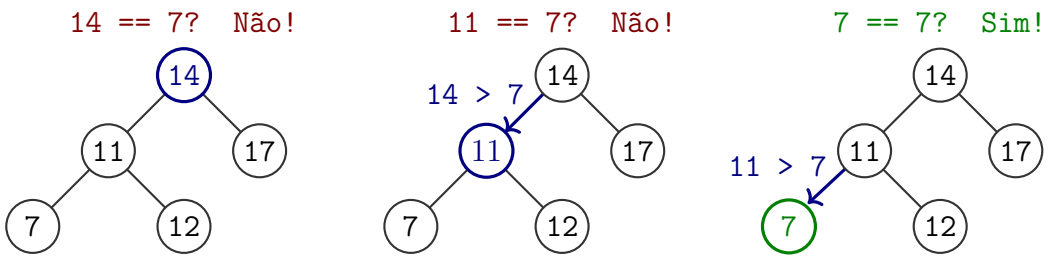


Figura 28 – Processo de busca do elemento 7. Note que a árvore não é nula e, dessa forma, o primeiro condicional é falso.

6.3.3.3 Inserção de Elemento

A inserção utiliza o mesmo conceito da busca. Na verdade, é realizado uma busca, porém o objetivo não é encontrar um valor específico, mas sim uma folha. Encontrado tal folha, alocamos memória para o novo nó e tornamos seu filho.

```
No* inserir(No *arv, int v) {
   if (arv == NULL) {
      arv = malloc (sizeof(No));
      arv->valor = v;
      arv->esq = arv->dir = NULL;
   } else if (arv->valor > v) arv->esq = inserir(arv->esq, v);
   else arv->dir = inserir(arv->dir,v);
   return arv;
}
```

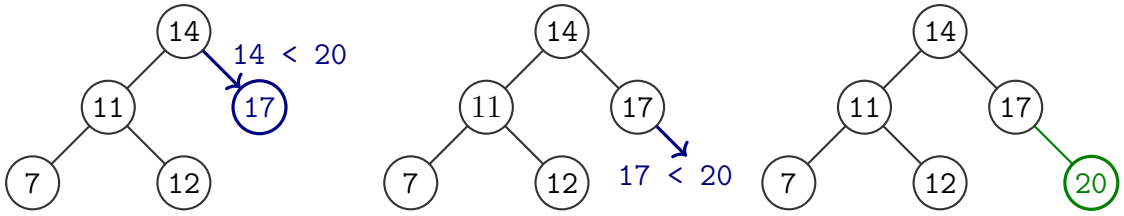


Figura 29 – Processo de inserção do elemento 20.

6.3.3.4 Remoção de Elemento

A remoção, como de costume, é a operação mais complicada de implementar. Nela, devemos observar três casos para excluir o nó.

1. **O nó é uma folha**: Nesse caso, basta removermos o nó da árvore.

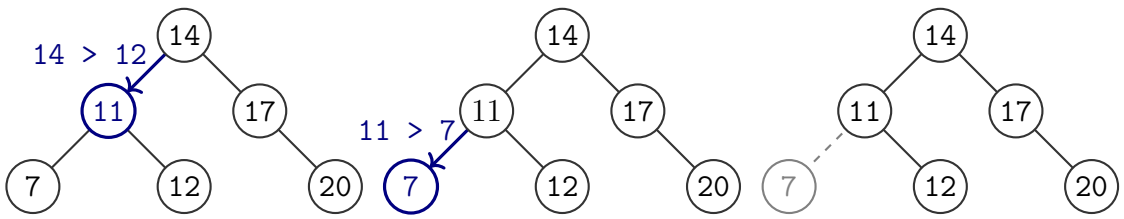


Figura 30 – Processo de remoção do elemento 7.

2. **O nó possui um filho**: Nesse caso, o seu filho tomará o lugar do pai, e este será excluído.

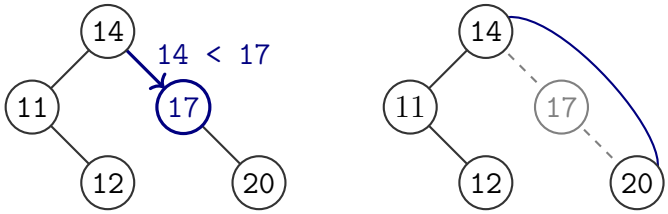


Figura 31 – Processo de remoção do elemento 17.

3. **O nó possui dois filhos**: Nesse caso, devemos encontrar o sucessor ideal para o pai, sendo este excluído. O substituto é o elemento mais a direita da subárvore esquerda ou o nó mais à esquerda da subárvore direita pois, dessa forma, respeitamos a ordenação da árvore. Será utilizada a primeira opção na implementação.

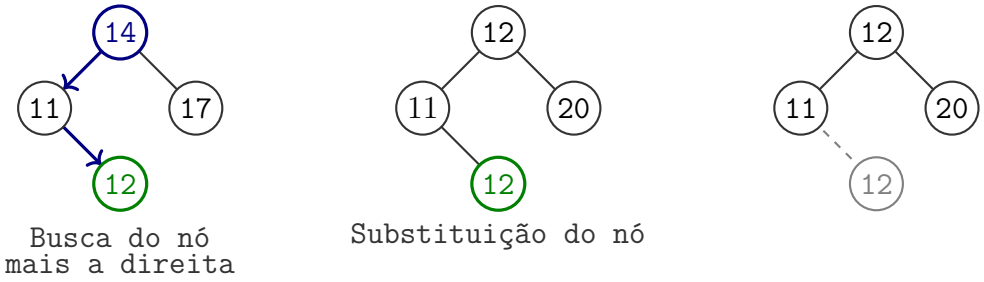


Figura 32 – Processo de inserção do elemento 14.

Inicialmente, é realizado uma busca do elemento que desejamos remover. Encontrado tal elemento, verificamos as condições estabelecidas anteriormente e, dependendo da condição satisfeita, a remoção é realizada.

```
No* retira (No* arv, int v) {
   if (arv == NULL) return NULL;
   else if (arv->valor > v) arv->esq = retira(arv->esq, v);
   else if (arv->valor < v) arv->dir = retira(arv->dir, v);
   else {
      if (arv->esq == NULL) {
         No* t = arv->dir;
         free (arv);
         return t;
      } else if (arv->dir == NULL) {
         No* t = arv->esq;
         free (arv);
         return t;
      } else {
         No* pai = arv;
```

```
16          No* f = arv->esq;
17          while (f->dir != NULL) {
18             pai = f;
19             f = f->dir;
20          }
21          arv->valor = f->valor;
22          f->valor = v;
23          arv->esq = retira(arv->esq,v);
24       }
25    }
26
27    return arv;
28 }
```

Note que, caso a remoção seja de uma folha, entramos na primeira condicional, retornando o filho direito, que também é vazio. Dessa forma, a primeira condição é satisfeita na segunda.

6.4 Percurso em árvores binárias

Em algumas situações, precisamos percorrer a árvore, visitando cada nó uma única vez. Por exemplo, quando queremos imprimir os elementos da árvore para teste. Como os dados não são organizados em sequência, podemos realizar várias formas de travessia, dependendo da aplicação, mudando a ordem em que os nós são visitados. Para efeitos práticos, veremos três tipos de percurso.

6.4.1 Percurso pré-ordem

Essa forma de percorrer uma árvore pode ser utilizada para criar uma cópia de uma árvore, pois visita o nó antes que seus filhos. O algoritmo para o percurso pré-ordem é dado por:

1. visita a raiz
2. recursivamente percorre a subárvore esquerda do nó
3. recursivamente percorre a subárvore direita do nó

Em C, temos:

```
1 void imprimirPreOrdem(No arv) {
2    if (arv == NULL) return;
3
```

```
4     printf("%i\n", arv->valor);
5     imprimirPreOrdem(arv->esq);
6     imprimirPreOrdem(arv->dir);
7  }
```

6.4.2 Percurso em ordem

Esse tipo de travessia também é conhecido como percurso em ordem simétrica. Em uma BST, esse percurso visita os nós em ordem crescente, podendo ser utilizado para imprimir os nós ordenado. Seu algoritmo é da seguinte forma:

1. recursivamente percorre a subárvore esquerda do nó
2. visita a raiz
3. recursivamente percorre a subárvore direita do nó

Em C, temos:

```
1  void imprimirEmOrdem(No arv) {
2     if (arv == NULL) return;
3
4     imprimirEmOrdem(arv->esq);
5     printf("%i\n", arv->valor);
6     imprimirEmOrdem(arv->dir);
7  }
```

6.4.3 Percurso pós-ordem

O percurso pós-ordem é utilizado para realizarmos a exclusão de uma árvore, quando desejamos liberar memória, deletando todos os seus nós. Seu uso é recomendado nessa situação pois, antes de excluir um nó, devemos excluir os seus filhos. O seu algoritmo é definido como:

1. recursivamente percorre a subárvore esquerda do nó
2. recursivamente percorre a subárvore direita do nó
3. visita a raiz

Em C, temos:

```c
void imprimirPosOrdem(No arv) {
    if (arv == NULL) return;

    imprimirPosOrdem(arv->esq);
    imprimirPosOrdem(arv->dir);
    printf("%i\n", arv->valor);
}
```

6.5 Árvores Genéricas

Com o uso das árvores binárias, limitamos a quantidade de filhos que um nó pode ter. Para isso, mudamos a definição de árvores inicial. Da mesma forma, poderíamos criar uma árvore que tenha, no máximo, três filhos (como as *tries* ternárias de busca), ou quatro, ou cinco e assim por diante.

Podemos ainda, para ser mais general, construir uma árvore que não possui limitações da quantidade de filhos, seguindo a definição dada na seção 6.1. Esse tipo de árvore é utilizada, por exemplo, em uma árvore de diretórios, onde uma pasta pode possuir diversas sub-pastas, assim como estes, da mesma forma, pode ter suas sub-pastas, e assim por diante, sendo todos sub-pastas da pasta raiz.

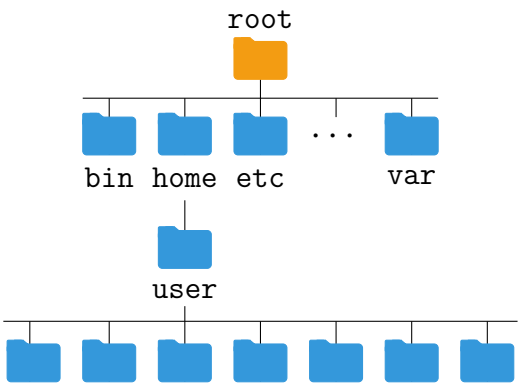


Figura 33 – Representação de uma estrutura de uma árvore de diretórios. Fonte: Autor.

6.6 Outras implementações de árvores

A árvore é um tipo de dado que possui muitas aplicações, uma vez que admite um tratamento computacional simples e eficiente[6]. Além disso, é bastante maleável em sua definição, havendo diversas versões de árvores com diferentes propriedades e complexidades. Iremos definir, de forma simplificada, algumas dessas estruturas para fim de conhecimento. Todas elas serão objeto de estudo para a disciplina de Estrutura de Dados Avançada.

Até então, utilizamos estruturas que armazenam um valor e que, de igual forma, é a chave de busca. Contudo, as estruturas que serão apresentadas normalmente utilizam um tipo para o valor que deseja armazenar e um outro tipo, que seja ordenável e comparável, para representar a chave. Como exemplo, podemos utilizar uma estrutura que armazene *arrays*, e tenha como chave um inteiro. Agora veremos tais estruturas.

6.6.1 Árvore 2-3

Uma árvore 2-3 é uma árvore que permite um ou dois elementos por nó, de forma que o nó com 1 elemento deve ter dois filhos e o nó com 2 elementos possui três filhos. Nessa estrutura, a quantidade de filhos é restrita ao que foi definido por essas propriedades, com exceção, obviamente, das folhas (não possui nenhum filho).

Essa estrutura possui um balanceamento perfeito, ou seja, todas as folhas estão no mesmo nível, ou melhor, todo caminho da raiz até as folhas tem o mesmo tamanho.

6.6.2 Árvore rubro-preta

Uma árvore rubro-negra é uma árvore binária de busca que possui um campo extra em sua estrutura, que armazena sua cor, sendo vermelha ou preta. Além disso, deve seguir as seguintes propriedades[2]:

1. a raiz é preta
2. toda folha é preta
3. se um nó é vermelho, seus filhos são pretos
4. Para cada nó, todos os caminhos desse nó às folhas descendentes contêm o mesmo número de nós pretos

Nessa estrutura, ao realizarmos as operações de inserção e remoção, em alguns casos, as suas propriedades são violadas. Para resolver isso, é necessário realizar mudanças de posições dos nós, por meio de rotações. Dessa forma, ao implemertamos a árvore rubro-negra, precisamos também implementar funções auxiliares que façam essas rotações.

6.6.3 Árvore AVL

Uma árvore AVL é uma árvore binária de busca onde, para todo nó, a profundidade da subárvore esquerda e da subárvore direita diferem no máximo em 1[2]. Ela é bastante utilizada em aplicações em que as buscas são mais frequentes que as inserções. Semelhante às árvore rubro-negra, ao implementarmos a AVL é essencial o uso das funções de rotações de nós.

6.6.4 Árvore B

Uma árvore B é uma árvore de busca m-ária, ou melhor, uma árvore de busca genérica. Ela é utilizada, por exemplo, em banco de dados e sistemas de arquivos. As árvores B são semelhantes às árvores rubro-negra, uma vez que possui uma altura logaritmíca (para toda árvore B com n nós, sua altura é $\log n$), porém permite mais de dois filhos, em contrapartida ao rubro-negra, podendo chegar em centenas ou até milhares[2].

Esse tipo de estrutura possui a vantagem de funcionar bem em memórias secundárias, de forma que realiza uma quantidade ínfima de operações de E/S no disco, comparada à outras estruturas.

6.6.5 *Tries*

As *tries*, também conhecidas como árvores digitais, são árvores utilizadas, normalmente, para utilizar strings como chaves. Nessa estrutura, cada nó armazena um caracter, com exceção da raiz, e seus filhos armazenam o próximo caracter de uma string. Tries que permitem r filhos são denominada R-Way Tries. Existem também, a trie ternária de busca, ou TST, que limita os nós à três filhos.

Podemos comparar a *trie* com um autômato finito determinístico, ainda que, usualmente não utilizamos a seta para representar a direção. Essa semelhança é vista quando relacionamos os nós como os estados do autômato e as ligações como as transições.

Exercício 26. *Em uma árvore binária, qual é o número máximo de nós que pode ser achado nos níveis 3, 4 e 12?*

Exercício 27. *Desenvolva uma função que receba uma árvore e um valor v. A função deve retornar 0 se não existe o valor v na árvore. Caso exista, a função deverá retornar quantas verificações foram realizadas.*

Exercício 28. *Prove, que uma árvore binária T com $n > 0$ nós, o número de subárvores vazias é n+1.*

Exercício 29. *Faça uma função que receba uma árvore e retorne a quantidade de nós dela.*

Exercício 30. *Faça uma função que receba uma árvore e retorne a altura dela.*

Exercício 31. *Faça uma função que receba uma árvore e retorne a quantidade de folhas dela.*

Exercício 32. *Desenvolva uma função que receba uma árvore e um valor v. A função deve retornar 0 se não existe o valor v na árvore. Caso exista, a função deverá retornar quantas verificações foram realizadas.*

Exercício 33. *Desenvolva a função de travessia que realiza visita por níveis da árvore. Esse modo de travessia é denominado percurso em largura. (Dica: será necessário utilizar uma fila).*

6.7 DESAFIO: Árvore genealógica

Uma árvore genealógica é uma forma de representar a história de uma família, a partir da descendência de um casal, sendo o patriarca e a matriarca. Devemos construir uma aplicação que receba um arquivo com uma árvore genealógica e, a partir dele, insira os nomes em uma árvore genérica. Para isso, é necessário construir uma estrutura de dados que armazene *strings* e que tenha um número indefinido de filhos (para tal, podemos utilizar uma lista encadeada, ou até mesmo uma árvore), mas, se prefirir, pode se limitar o total de filhos que uma pessoa pode ter.

Em relação ao arquivo, ele deve ter uma estrutura que defina os vínculos de cada pessoa. Para isso, pode ser utilizado uma representação com parênteses aninhados (veja a seção 6.1). Mas isso é critério individual. Vale lembrar que é necessário construir uma função que receba o arquivo e "converta" para a estrutura que for construída.

Ademais, é importante salientar que deve ser construido todas as operações básicas (inserção, remoção, busca e travessia) para a aplicação, além de operações extras, dependendo da forma em que for construída a árvore.

7 Busca binária

A busca binária é um importante algoritmo para realizar buscas em um conjunto ordenado de elementos. Para exemplificar, inicialmente, considere um *array* A com n elementos que armazena inteiros dispostos em ordem crescente.

k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	$\cdots$	k_{n-2}	k_{n-1}	k_n

Figura 34 – Representação de um vetor, onde $k_i \leq k_i + 1, \forall i < n$.

Agora, suponha que estamos buscando um elemento e com valor j em A. Para realizar a busca, devemos verificar cada posição até encontrar o que buscamos, se $e \in A$. Para isso, podemos realizar uma busca sequencial, onde utilizamos um laço de repetição, partindo da posição 0 do *array*, e visitamos cada posição até encontrar o valor buscado. Esse tipo de busca é dito linear.

Evidentemente, com esse algoritmo, realizamos a busca do elemento e, caso pertença ao *array*, será encontrado. Porém, essa solução possui um custo linear, que pode ser desfávoravel. Isso porque, em uma busca no *array* A, teremos que realizar n verificações,

no pior caso. E qual seria o pior caso? Note que, se j é o maior valor armazenado no *array*, precisamos percorrer todo o *array* para o encontrarmos. Da mesma forma ocorre se $e \notin A$.

Para *arrays* com poucos elementos, isso não interfere tanto no tempo de execução do algoritmo. Porém, para *arrays* com grande quantidade de elementos, isso faz diferença. Como exemplo, considere um *array* que armazene 2^{30} inteiros, ordenados (uma curiosidade: em C, um inteiro possui 4 *bytes* e, com isso, este *array* utiliza 4GB!). No pior caso, teríamos que verificar 2^{30} elemento, o que não seria interessante.

Portanto, é interessante buscarmos outra forma de realizarmos essa busca. Como A é ordenado, temos que $k_{i-1} < k_i \leq k_{i+1}$. Com isso, sabemos que todo elemento antes de k_i é menor que k_i e que todo elemento depois, é maior. Por conseguinte, se buscamos o elemento e, sabendo que $j \leq k_i$, não precisamos verificar os elementos maiores que j, uma vez que sabemos que é menor. Dessa maneira, evitamos verificações desnecessárias.

E é nisso que a busca binária se baseia. É feita sucessivas divisões no espaço de busca comparando o elemento buscado com o elemento central. Dessa forma, eliminamos sempre metade do espaço de busca. Utilizando a busca binária em um *array* com 2^{30} elementos, reduzimos as verificações para no máximo 30. Isso é um ganho considerável de performance!

Algoritmo 6 Realiza uma busca binária em um *array*.

```
função BuscaBinaria(A[0..n], valor)
    esq ← 0
    dir ← n − 1
    enquanto esq ≤ dir faça
        metade ← ⌊(esq + dir)/2⌋
        se valor = A[metade] então
            retorne metade
        fim se
        se valor < A[metade] então
            sup ← meio − 1
        senão
            inf ← meio + 1
        fim se
    fim enquanto
    retorne Sem Sucesso
fim função
```

Observe que, até então, utilizamos um *array* para armazenar os dados mas, manter um *array* ordenado é complicado, uma vez que, em toda inserção/remoção, devemos reordená-lo. Dessa forma, precisamos de uma estrutura que seja ordenado dinamicamente.

Das estruturas estudadas, utilizar uma lista, pilha ou fila não é recomendado uma vez que não temos acesso para todas as posições, de forma eficiente. Não obstante, as

árvores, dependendo da forma que for definida, nos pode ser útil. Na verdade, estudamos um tipo de árvore que nos permite realizar uma busca binária: a árvore binária de busca.

Sendo assim, a busca implementada na BST é, basicamente, uma busca binária pois, eliminamos da busca uma parte dos nós da árvore dependendo se o valor comparado é maior ou menor que o valor do nó. Dessa maneira, com uso da BST não precisamos se preocupar em manter o conjunto ordenado, uma vez que sabemos que ele já estar ordenado, pela definição da BST.

Exercício 34. *Porque a busca binária possui um custo logarítmico?*

8 Algoritmos de ordenação

A ordenação de elementos é bastante utilizado no nosso dia a dia. Por exemplo, a busca de uma informação no dicionário é mais simples quando estão organizados em ordem alfabética, assim como na pesquisa de um contato em uma lista telefônica, ou até mesmo ao buscarmos um arquivo no computador. Ordenar elementos é rearranjá-lo em uma ordem. Podemos ordenar qualquer informação, desde que exista comparabilidade entre os elementos do conjunto.

Na busca binária, como estudado no tópico anterior, é preciso que os elementos da estrutura estejam ordenados de forma crescente ou decrescente, porém, nem sempre temos um conjunto de elementos ordenados. Na verdade, dificilmente temos um conjunto bem ordenado, dado como entrada. Com isso, devemos buscar maneiras de manter a ordenação dos dados para que esse algoritmo funcione.

Podemos manter a ordenação dos dados de duas formas: garantindo que os dados, no momento da inserção, respeitem a ordenação estabelecida; ou ordenando o conjunto de dados, já criado, com um algoritmo de ordenação[4].

Na primeira opção, temos que limitar as entradas do conjunto para apenas aqueles que respeitem a classificação. Por exemplo, dado um *array* A, com $n-1$ elementos, na inserção do n-ésimo elemento temos que o valor v_n deve ser maior que v_{n-1}, considerando uma ordenação ascedente. Com a segunda opção, devemos, com o conjunto já criado, ordená-lo, alternando as posições de cada elemento. E é isto que estudaremos nessa seção!

Existe diversos algoritmos que realizam tal operação. Cada um possui uma técnica própria e, dependendo da aplicação, possui suas vantagem e desvantagens. Para simplificar as operações, será utilizado *arrays* que armazenam inteiros em ordem crescente. Vamos apresentar os algoritmos mais famosos e mais estudados.

8.1 Insertion sort

Neste método de ordenação simples, cada elemento do *array* é retirado de sua posição e "inserido" na correta. Um exemplo clássico (e bem didático) para explicar essa técnica é o uso de um baralho de cartas. Normalmente, um pessoa que está jogando pôquer (ou qualquer outro jogo que use cartas) organiza suas cartas em ordem crescente, da esquerda para a direita.

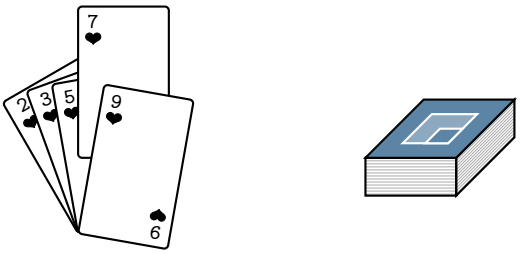


Figura 35 – No baralho, ordenamos as cartas de forma ascendente. Para cada carta que é puxada da mesa, inserimos na posição correta das cartas na mão. Fonte: Autor.

Com um *array* de inteiros, a ideia é a mesma. Percorremos o vetor, da esquerda para a direita, e, para cada elemento (carta), selecionamos e inserimos na posição correta à esquerda, sendo o subarray já ordenado (adicionamos nas cartas na mão) na sua posição correta. A figura a seguir, ilustra esse método de ordenação em um *array* de inteiros.

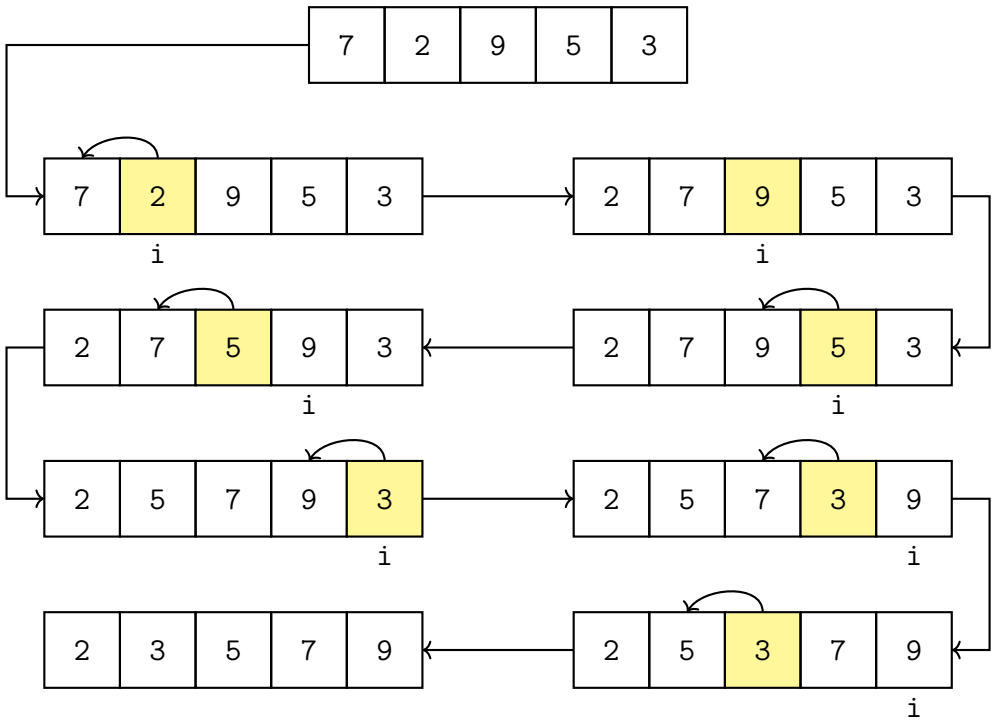


Figura 36 – Representação de um *array* de inteiros. O índice i representa a posição da iteração. Em amarelo, temos o valor que será trocado de posição.

Note que iniciamos a ordenação na segunda posição do *array*. Isso ocorre pois, quando consideramos um vetor unitário, sabemos que ele já está ordenado.

8.1.1 Implementação

O algoritmo do insertion sort segue o mesmo princípio do que foi exemplicado na Figura 36. É utilizado um laço de repetição que percorre o array e, a partir de uma variável auxiliar, é realizado permuta de posições, onde move-se o elemento para a esquerda, quando necessário. Segue o algoritmo:

Algoritmo 7 Algoritmo de ordenação por inserção.

```
função INSERTIONSORT(A[0..n])
    para i ← 1 até n − 1 faça
        chave ← A[i]
        j ← i − 1
        enquanto j ≥ 0 e A[j] > chave faça
            A[j + 1] ← A[j]
            j ← j − 1
        fim enquanto
        A[j + 1] ← chave
    fim para
fim função
```

A váriavel i é a responsável por percorrer o array e a váriavel j, representa as possíveis trocas de um elemento. Note que j é definido inicialmente na posição anterior de i, e é verificado se a posição j possui um valor maior que o elemento que será adicionado na sua posição correta (representado pela variável auxiliar *chave*).

Exercício 35. *Faça uma função que ordene um vetor em ordem decrescente, com o algoritmo de ordenação insertion sort.*

Exercício 36. *Desenvolva uma versão recursiva do algoritmo insertion sort.*

Exercício 37. *Faça um função que leia n strings e ordene-as pelo tamanho.*

Exercício 38. *Crie um função que dado uma string, coloque as letras dela em ordem decrescente, pelo insertion sort.*

8.1.2 Análise do insertion sort

Observe que, ao inserirmos um elemento no subarray ordenado, provavelmente será necessário realizar trocas. Por exemplo, ao inserir o valor 1 no array da figura 36, devemos mover todos os elementos do vetor para que o elemento inserido em sua posição correta.

Quando consideramos um array, de tamanho n, em ordem inversa ao da classificação do algoritmo (por exemplo, um vetor ordenado decrescente como entrada do insertion sort que ordena crescente), em toda iteração o elemento a ser adicionado será inserido no início do subarray. Dessa forma, na 1ª iteração, temos que realizar uma troca de posição, na 2ª, duas trocas, na 3ª, três, e na (n-1)-ésima, $n-1$ trocas. Com isso, temos $\sum_{i=1}^{n-1} i$ trocas e, dessa forma, um custo quadrático.

Ainda assim, o insertion sort é útil para estruturas lineares pequenas, tendo como principal vantagem o número pequeno de comparações realizadas e sua desvantagem, como visto anteriormente, é o número excessivo de trocas[8].

8.2 Selection sort

O algoritmo de ordenação selection sort é extremamente simples. Ele percorre o array e seleciona o menor (ordem crescente) ou maior (ordem decrescente) elemento e coloca na primeira posição, depois seleciona o segundo menor/maior e coloca na segunda posição e assim por diante.

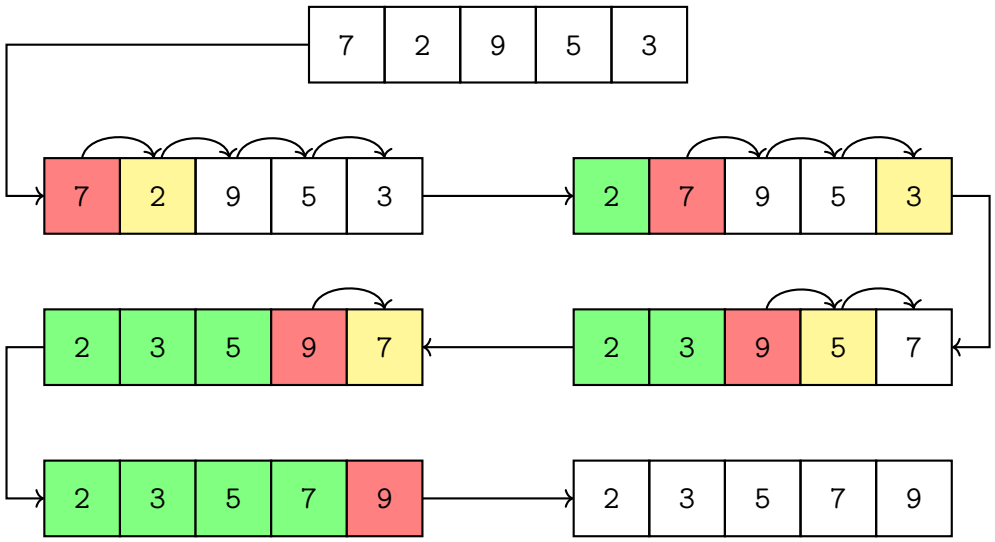


Figura 37 – Representação da ordenação de um *array* de inteiros. Em vermelho, temos a posição da iteração, de forma que é o valor comparado com os elementos posteriores; em amarelo, a posição que será trocada; e em verde o subarray que está ordenado

8.2.1 Implementação

O algoritmo do selection sort lembra bastante o do insertion sort. Ele possui dois laços de repetição: um percorre o vetor, sendo o elemento da posição comparado com as posteriores; o outro, interno ao primeiro, é responsável por procurar o menor elemento, em comparação ao da iteração do primeiro laço. Segue o algoritmo do selection sort:

Algoritmo 8 Algoritmo de ordenação por seleção.

```
função selectionSort(A[0..n])
    para i ← 0 até n − 1 faça
        k ← i
        j ← i + 1
        enquanto j < n faça
            se A[j] < A[k] então
                k ← j
            fim se
            j ← j + 1
        fim enquanto
        trocarPosicao(A[i], A[k])
    fim para
fim função
```

Exercício 39. *Faça uma função que dado uma string, coloque os seus caracteres em ordem crescente.*

Exercício 40. *Desenvolva uma versão recursiva do algoritmo selection sort.*

Exercício 41. *Modifique o selection sort, de tal forma que retorne o número de trocas realizadas. Crie outra versão que retorne o número de comparações.*

8.2.2 Análise do selection sort

No insertion sort, para cada elemento, é realizado uma comparação com todos os outros. Por exemplo, em um array com n elementos, temos que o 1º elemento realiza $n-1$ comparações, o 2º, $n-2$ comparações e o n-ésimo não realiza nenhuma, dessa forma, temos no total

$$(n-1)+(n-2)+(n-3)+\cdots+1+0=\sum_{i=0}^{n-1} i=\frac{n^2+n}{2}$$

comparações. Com isso, temos um custo quadrático para as comparações para qualquer entrada.

Ainda assim, o selection sort possui a vantagem de realizar muito menos trocas do que comparações, ao contrário do insertion sort. Dessa forma, ele pode ser utilizado "quando se trabalha com componentes em que, quanto mais se escreve, ou reescreve, mais se desgasta, e, consequentemente, perdem sua eficiência, como é o caso das memórias EEPROM e FLASH"[8].

8.3 Bubble sort

Este algoritmo realiza a ordenação comparando dois elementos e levando o maior elemento a direita. O seu nome é dado por uma associação de que os elementos maiores

(ou menores, dependendo da ordem) são mais leves, e flutuam como bolhas até suas posições corretas[4].

Por exemplo, considere um array com n elementos que queremos ordenar de forma crescente, o elemento e_1 é comparado com o e_2 e se $e_1 < e_2$, estão invertemos as posições, caso $e_1 \geq e_2$, as posições são mantidas. Depois, comparamos e_2 com e_3, e após estes e_3 com e_4 até $e_{(n-1)}$ com e_n, realizando as trocas necessárias. Isso é realizado até que o maior elemento seja levado para a última posição. O processo continua levando o segundo maior, depois o terceiro e assim por diante.

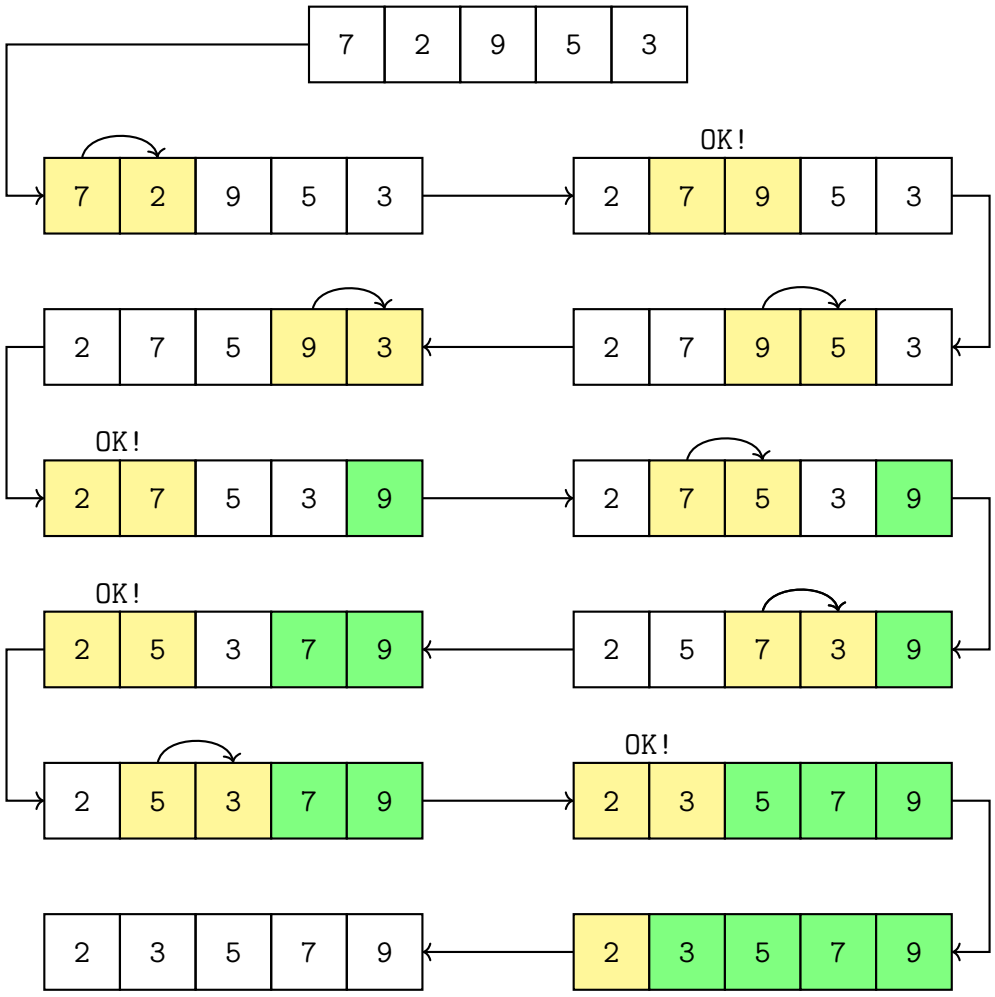


Figura 38 – Representação da ordenação de um *array* de inteiros. Em amarelo, temos os elementos que estão sendo comparado na iteração e em verde o subarray ordenado.

8.3.1 Implementação

A implementação desse algoritmo é bastante simples. Ela possui dois laço: um responsável por percorrer todo o array; e outro, interno ao primeiro, é encarregado por realizar as verificações e permutas necesárias, levando o maior elemento do subarray para o final. No laço mais interno, é desconsiderado a parte final do array que já está ordenado,

devido a iteração anterior. Com isso temos duas regiões no array A: a primeira, $A[0..n-i-1]$, que precisa ser ordenado (n é o tamanho do array e i o índice da iteração); e o segunda, $A[n-i..n]$, já ordenado (na figura 38, é representada pela cor verde).

Algoritmo 9 Algoritmo de ordenação bolha.

```
função BUBBLESORT(A[0..n])
    para i ← 0 até n − 1 faça
        para j ← 0 até n − i − 1 faça
            se A[j] < A[j + 1] então
                TROCARPOSICAO(A[j], A[j + 1])
            fim se
        fim para
    fim para
fim função
```

Exercício 42. *Explique como seria possível melhorar o método bubblesort, armazenando não apenas a informação da troca, mas também a posição do vetor onde ocorreu a troca. Implemente essa modificação.*

Exercício 43. *Desenvolva uma versão recursiva do algoritmo bubble sort.*

Exercício 44. *Crie uma estrutura* Aluno*, que armazena a matrícula e o nome, e desenvolva uma função que recebe um array de alunos e ordene em ordem crescente, com o bubble sort.*

8.3.2 Análise do bubble sort

Assim como nos algoritmos anteriores, usaremos o número de comparações para definir seu custo. Da mesma forma, dado um array com n posições, em cada iteração é realizado $n-i$ comparações, onde i é a posição atual do iterador, de forma que $i \leq n-1$. No total, temos $\frac{(n^2+n)}{2}$ comparações, ou seja, um custo quadrático.

8.4 Mergesort

Depois de apresentar métodos simples de ordenaçãos, e de certa forma ineficiente em geral, agora iremos estudar estruturas mais complexas, porém eficientes.

Iniciaremos com mergesort, um algoritmo que se baseia no paradigma de divisão e conquista. Nessa técnica, o algoritmo divide o problema em versões menores, para resolvê-los, construíndo, a partir da solução dos menores, a solução do problema inicial (essa técnica de programação será estudada em Projeto e Análise de Algoritmos).

Dessa forma, o mergesort particiona o *array* em dois subarrays, ordena-os e, no fim, realiza a mesclagem dos subarrrays ordenados, de forma que completa a ordenação, sendo

realizado recursivamente. Segue o procedimento de dividir e conquistar do mergesort para um *array* A[5]:

Dividir: o *array* A é dividido em dois subarrays, A_1 e A_2, onde A_1 contém os primeiros $\left\lceil \frac{n}{2} \right\rceil$ de A e A_2 possui os $\left\lfloor \frac{n}{2} \right\rfloor$ elementos restantes.

Conquistar: Os subarrays A_1 e A_2 são ordenados recursivamente, por meio de chamadas do mergesort.

Combinar: Os subarrays A_1 e A_2, já ordenados, são mesclados de forma que torna-se um único array ordenado.

O uso da palavra mesclar, no processo de combinar os subarrays, é bem utilizado pois devemos "misturá-los", para tornar um todo também ordenado. Diferente do quicksort, onde se utiliza uma função auxiliar antes das chamadas recursivas, no mergesort, o vetor é dividido até o caso base, isto é, até temos um subarray com um único elemento ou com nenhum elemento, para depois usar a função auxiliar.

Com os subarrays ordenado, podemos realizar a combinação, referente a chamada recursiva. Essa mesclagem é dado utilizando um vetor auxiliar, que possui um tamanho equivalente à soma da quantidade de A_1 e A_2. Com isso, intercalamos os subarrays, inserido, no vetor auxiliar, os elementos de menor valor (ou maior, para ordenação decrescente).

8.4.1 Implementação

Basicamente, devemos realizar duas chamadas recursivas do mergesort, para as duas metades do array e, no fim, unir os subarrays produzidos. Para tal, precisamos da posição central do array, para orientar a função de mesclagem.

Algoritmo 10 Algoritmo da operação mergesort.

```
função MERGESORT(A[p..r])
    se p < r então
        m ← ⌊(n/2)⌋
        MERGESORT(A[p..m])
        MERGESORT(A[p..m + 1])
        MERGE(A[p..r], m)
    fim se
fim função
```

A função *merge*, apesar de parecer complexo, é bastante simples e intuitiva. Basicamente é realizado uma verificação que intercala os dois subarrays, comparando um elemento do primeiro com um do segundo. E então, é inserido o elemento menor ou igual

em um vetor auxiliar (como explicado anteriormente). Depois disso, é inserido os elementos remanescente de um dos subarrays e então o *array* original torna-se o vetor auxiliar. Na implementação a divisão do array é dado por um índice que indica o índice central.

Algoritmo 11 Algoritmo que mescla dois arrays ordenados.

```
função MERGE(A[p..r], m)
    i ← p
    j ← m + 1
    k ← 0
    enquanto i ≤ m e j ≤ r faça
        se A[i] ≤ A[j] então
            B[k] ← i
            i ← i + 1
        senão
            B[k] ← A[j]
            j ← j + 1
        fim se
        k ← k + 1
    fim enquanto

    enquanto i ≤ m faça
        B[k] ← A[i]
        i ← i + 1
        k ← k + 1
    fim enquanto

    enquanto j ≤ r faça
        B[k] ← A[j]
        j ← j + 1
        k ← k + 1
    fim enquanto

    A ← B
fim função
```

Exercício 45. *Dados três vetores ordenados, implemente uma função que intercala e retorne o vetor resultante ordenado. Implemente uma função merge3_sort, que faça ordenação em um vetor utilizando a sua função de intercalação (sugestão: se baseie no algoritmo do merge-sort original). Qual a complexidade desse algoritmo?*

8.4.2 Análise do mergesort

O mergesort é um algoritmo que possui uma complexidade linearítmica, para todos os casos. Isso o torna muito eficiente comparado aos métodos mais simples. Porém, ele apresenta a desvantagem de precisar armazenar um espaço extra na memória, uma vez que

precisamos de um vetor auxiliar, para cada chamada recursiva, ocasionando um consumo de espaço total de $O(n \log n)$.

8.5 Quicksort

Este algoritmo, assim como o mergesort, utiliza a estratégia de divisão e conquista, que divide o *array* em dois subarrays, baseando-se em um dos seus elementos, e a partir desse elemento realiza a "conquista". O quicksort, com frequência, é considerada a melhor opção prática para ordenação[2]. Segue o processo de dividir e conquistar, no quicksort, para um vetor $A[p..r]$[2]:

Dividir: O *array* $A[p..r]$ é particionado em dois subarrays $A[p..q-1]$ e $A[q+1..r]$, onde cada elemento de $A[p..q-1]$ é menor ou igual que $A[q]$, e todo elemento de $A[q+1..r]$ é maior que $A[q]$. O índice q é calculado como parte do particionamento.

Conquistar: Os subarrays $A[p..q-1]$ e $A[q+1..r]$ são ordenados por meio de chamadas recursivas do quicksort.

Combinar: Como os subarrays são ordenados localmente, esse passo é irrelevante. No fim, o array A[p..r] já está ordenado.

O elemento $A[q]$ é o ponto de partida da divisão, sendo denominado pivô. Podemos escolher qualquer elemento para ser o pivô, porém isso pode interferir no custo do algoritmo. Usaremos como pivô o elemento da posição r, ou seja, o último elemento.

8.5.1 Implementação

O quicksort é implementado recursivamente, da mesma forma que foi descrito anteriormente, nos moldes da divisão e conquista, de modo que, primeiro particionamos o array e depois, realizamos as chamadas recursivas para os dois subarrays.

Algoritmo 12 Algoritmo da operação quicksort.

```
função QUICKSORT(A[p..r])
    se p < r então
        q ← PARTITION(A[p..r])
        QUICKSORT(A[p..q − 1])
        QUICKSORT(A[q + 1..r])
    fim se
fim função
```

Feito a função base, podemos implementar o responsável por reorganizar o subarray $A[p..r]$, de forma que os maiores ficam após o pivô e os menores ou iguais, antes. Definimos o pivô x como $A[r]$.

Vale salientar que, à medida que o procedimento é executado, o vetor é dividido em quatro grupos, em ordem: os elementos menores ou iguais à x; os elementos maiores que x; os elementos que ainda não foram reorganizados; e um grupo unitário, constítuido pelo próprio x.

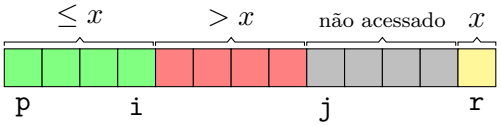


Figura 39 – Exemplo da distribuição das regiões ou grupos do array durante o processo do particionamento. O objetivo é tornar a região cinza vazia.

Após isso, precisamos de um índice que aponte para o último elemento, dos menores ou iguais a x, representando o primeiro grupo, que denominamos i. Depois, percorremos o *array*, para verificar se o elemento $e_j \leq x$, onde $j \leq r - 1$. Caso seja, devemos incrementar i e permutar e_j com e_i, isto significa que adicionamos um elemento ao primeiro grupo. Caso não seja, continuamos o percurso. No fim, temos que o terceiro grupo é vazio, e podemos posicionar x em sua posição. Para isso, basta trocarmos de posição x com o primeiro elemento do segundo grupo.

Algoritmo 13 Algoritmo que reorganiza o array, particionando em dois grupos que se baseiam no pivô.

```
função PARTITION(A[p..r])
    x ← A[r]
    i ← p − 1
    para j ← p até r − 1 faça
        se A[j] ≤ x então
            i ← i + 1
            TROCARPOSICAO(A[i], A[j])
        fim se
    fim para
    TROCARPOSICAO(A[i + 1], A[r])
    retorne i + 1
fim função
```

Exercício 46. *Reescreva o procedimento de partição do QuickSort tomando como referência (pivô) o primeiro elemento.*

Exercício 47. *Crie um programa que dado uma string, coloque suas letras em ordem crescente pelo algoritmo quicksort.*

Exercício 48. *Desenvolva uma versão iterativa do Quicksort.*

Este algoritmo utiliza a partição proposta por Nico Lomuto, popularizada por Cormen et al., sendo uma versão muito simples e didática. Não obstante, existe outro

algoritmo de partição, criado por C.A.R. Hoare, sendo este mais eficiente, uma vez que realiza menos trocas. Vale salientar que Hoare é o criador do quicksort, sendo a sua versão de partição a originalmente descrita.

Na partição de Hoare, é utilizado dois índices, que representam as extremidades do *array*. Estes índices se movem um em direção ao outro, até encontrar um elemento maior e outro menor ou igual em ordem invertida. Neste caso, os elementos são permutados e o processo é continuado até uma extremidade encontrar a outra. Definimos o pivô x como $A[p]$.

Algoritmo 14 Algoritmo que realiza a partição do array, utilizando o método de Hoare.

```
função PARTITION(A[p..r])
    x ← A[p]
    i ← p − 1
    j ← r + 1
    enquanto TRUE faça
        repita
            i ← i + 1
        até A[i] < x
        repita
            j ← j − 1
        até A[j] > x
        se i ≥ j então
            retorne j
        fim se
        TROCARPOSIÇÃO(A[i], A[j])
    fim enquanto
fim função
```

8.5.2 Análise do quicksort

O custo do algoritmo quicksort dependerá de como será realizado o particionamento que, por sua vez, depende da escolha do pivô. Se considerarmos um array ordenado, a escolha do último elemento como pivô não é o ideal, uma vez que selecionamos o maior (ou menor, dependendo da classificação estabelecida) do array e, dessa forma, o partição produzirá um subarray com n-1 elementos e outro com 0 elementos. Isso gerará um custo quadrático, sendo este o pior caso do quicksort. Porém, este caso dificilmente ocorre e, caso ocorra, podemos realizar uma simples verificação se o vetor está ordenado, o que tem um custo linear.

O melhor caso do quicksort ocorre quando consideramos a mediana do array como pivô. Dessa forma, temos um particionamento balanceado, onde um subarray possui $\left\lfloor \frac{n}{2} \right\rfloor$ elementos, e o outro $\left\lceil \frac{n}{2} - 1 \right\rceil$. Isso gerará um custo linearítmico, ou seja, $O(n \log n)$.

Entretanto, não é uma tarefa fácil para conseguirmos a mediana do array, mantendo essa complexidade.

Em geral, deseja-se aproximar o algoritmo para o melhor caso a partir de um pivô que seja próximo da mediana. Ou seja, estamos buscando encontrar elementos que estejam no "meio" do array ordenado. Escolhendo um elemento desse grupo central, teremos um custo que é assintoticamente o mesmo do melhor caso.

Para tal, é introduzido o conceito de randomização no algoritmo, onde a escolha um pivô é dada aleatoriamente. Baseando-se nesse conjunto desejado, temos uma maior chance de alcança-lo, realizando divisões razoalvementes balanceadas. Dessa forma, o caso médio é mais próximo do melhor caso.

Por fim, ainda que as próximas estruturas que serão estudadas tenham um custo linearítmico para todos os casos, o quicksort é mais rápido na prática, de forma que se apresenta uma boa opção para situações em que o objetivo é a execução em um menor tempo[8].

Exercício 49. *Faça uma versão randomizada do quicksort.*

8.6 Heapsort

Este algoritmo de ordenação utiliza de uma estrutura de dados em sua construção: a heap binária. Essa estrutura é usada para ordenar os elementos, enquanto eles são inseridos para que, no fim, os elemento serem removidos sucessivamente da raiz da heap, mantendo a sua propriedade.

8.6.1 Heap

Antes de partimos para o heapsort, é importante entendermos o funcionamento de uma heap. Uma heap binária, é um tipo de árvore binária completa, onde os nós representam os elementos e cada nó possui dois filhos, que são dois elementos descendentes ao pai (se necessário, revise a seção 6).

Como iremos implementar um heapsort utilizando *arrays*, precisamos definir a estrutura da árvore no vetor. Seja A um array de inteiros, com n elementos. A raiz do árvore é $A[0]$ e os seus filhos serão os elementos $A[1]$ e $A[2]$, os filhos de $A[1]$ serão os elementos $A[3]$ e $A[4]$, e os filhos de $A[i]$ serão os elementos $A[2i+1]$ e $A[2i+2]$. Dessa forma, temos acesso ao filho esquerdo a partir do índice $2i+1$ e ao filho direito pelo índice $2i+2$. A Figura 40, exemplica a organização dos elementos do *array* e do seus respectivos índices.

Agora, vamos definir o que é um heap. Existe dois tipos de heap: o máximo e o mínimo. No heap máximo, o valor do nó é sempre menor ou igual ao pai, e este é maior

ou igual ao seus filhos. No heap mínimo, é o inverso: os filhos de um nó é sempre maior ou igual, e seu pai é sempre menor ou igual.

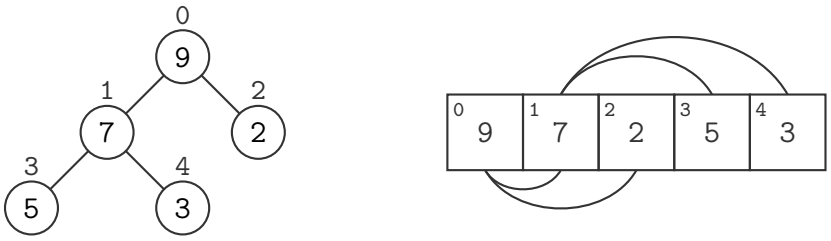


Figura 40 – À esquerda a representação de uma heap por uma árvore binária, e a direita por meio de um vetor. Neste exemplo, estamos utilizando um heap máximo.

Como estamos lidamos com um array e os filhos de um nó serão seus sucessores, a inserção de elementos é realizada pelos os níveis da árvore, da esquerda para a direita. Para o heapsort, vamos utilizar um heap máximo.

8.6.2 Implementação

Antes de implementar a heapsort, precisamos de duas funções auxiliares para manter a propriedade da heap e para construi-lo. Primeiramente, devemos descrever uma função que mantenha a propriedade da heap, nomeamos *heapify*.

Essa função recebe um *array* A e um índice i, onde os filhos de $A[i]$ são necessariamente heaps máximos, porém $A[i]$ pode ser menor que seus filhos, de forma que viole a propriedade do heap máximo. Então, nosso objetivo é mudar a posição de $A[i]$ para que esse subarray torne-se um heap máximo, "descendo" o elemento $A[i]$.

Algoritmo 15 Algoritmo que transforma um array "quase" heap, em uma heap.

```
função HEAPIFY(A, i)
    l ← 2 · i + 1
    l ← 2 · i + 2
    m ← i
    se l < |A| e A[l] > A[m] então
        m ← l
    fim se
    se r < |A| e A[r] > A[m] então
        m ← r
    fim se
    se m ≠ i então
        TROCARPOSICAO(A[i], A[m])
        HEAPIFY(A, m)
    fim se
fim função
```

Agora, vamos contruir um procedimento que transforma um array $A[0..n]$, onde $n = |A|$, em um heap máximo. Para isso, basta utilizarmos um laço que faça chamadas da função *heapify*.

Algoritmo 16 Algoritmo que torna um array qualquer em um heap máximo.

```
função BUILDHEAP(A)
    n ← |A|
    para i ← ⌊n/2⌋ até n faça
        HEAPIFY(A, i)
    fim para
fim função
```

Na figura 40, temos a representação de um heap máximo. Heap mínimo pode ser utilizado para implementar uma fila de prioridade. Por fim, podemos implementar o heapsort. Resumidamente o heapsort constrói o heap, remove a raiz, reconstrói o heap, remove a nova raiz e repete a operação até remover todos os elementos. Como a raiz é sempre o maior elemento, inserimo-a do fim do array para o ínicio.

Algoritmo 17 Algoritmo que torna um array qualquer em um heap máximo.

```
função HEAPSORT(A)
    BUILDHEAP(A)
    n ← |A|
    para i ← n − 1 até 0 faça
        HEAPIFY(A[..i], 0)
    fim para
fim função
```

Exercício 50. *Da forma que definimos o array para o heap, não temos acesso fácil para o pai de um nó. Porém, existe uma forma que definimos os índices de forma que os temos acesso ao pai de um nó em um tempo constante, assim como aos filhos. Qual seria esta forma e como realizamos tal mudança?*

Exercício 51. *Mostre que uma heap de n elementos tem altura* $\lfloor \log n \rfloor$.

Exercício 52. *Um array ordenado de forma crescente é um heap mínimo?*

Exercício 53. *Mostre que, em um array com n elementos, as folhas da heap são indexados por* $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, ..., n$.

8.6.3 Análise do heapsort

No heapsort, um número constante de elementos do *array* é armazenado fora do array de entrada. Isso significa que não é utilizado muito espaço em sua execução. Assim como os dois algoritmos anteriores, o heapsort possui um custo $O(n \log n)$.

Dos procedimentos implementados, a *heapify* possui um custo logarítmico, uma vez que realizamos uma descida na árvore. A *buildHeap* possui um custo linear e, como é feita $n-1$ chamadas do *heapify*, chegamos ao custo apresentado.

9 Grafos

A teoria dos grafos é uma área de estudo da matemática de suma importância, dispondo de diversas aplicações. Por exemplo, como distribuir n empregos para n pessoas com a melhor eficiência? Como distribuir os caminhos de fios de um chip para que evite cruzamentos? Qual melhor rota entre duas cidades? Com os grafos, podemos formalizar problemas que possuem um conjunto de elementos e eles se relacionam entre si.

O ponto de partida para o estudo dos grafos surgiu a partir de um problema bastante famoso, resolvido por Leonhard Euler, matemático suiço, conhecido como as pontes de *Königsberg*. *Königsberg* é uma cidade localizada na Prússia, cortada por um rio, onde possuía duas ilhas que são conectadas por com sete pontes. O problema é: existe uma maneira de atravessar todos as pontes, sem repeti-las? Euler utilizou um conceito que futuramente se tornaria os grafos.

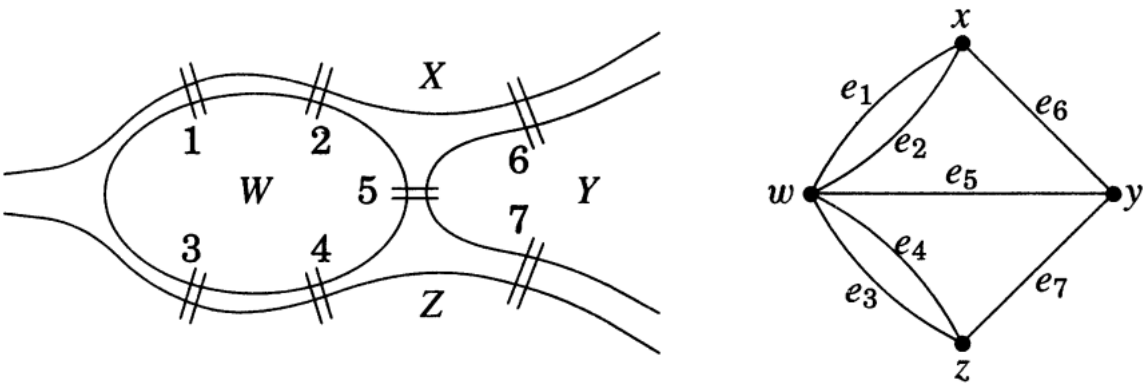


Figura 41 – Representação das sete pontes de *Königsberg* À direita, temos uma forma de apresentá-lo de uma forma mais simples. Fonte: Livro Introduction to Graph Theory - Douglas B. West[9].

Formalmente, definimos um grafo G como uma tripla ordenada que consiste em um conjunto de vértices $V(G)$, um conjunto de arestas $E(G)$, e uma relação que associa cada aresta com dois vértices, não necessariamente distintos[9]. Para representar um grafo, cada ponto indica um vértices, e as curvas as arestas, de forma que elas ligam dois pontos (vértices). Os vértices que são conectados é dita extremidade da aresta correspondente.

Por exemplo, considere o conjunto de vértices V=$\{v_1, v_2, v_3, v_4, v_5\}$, e de arestas E=$\{e_1, e_2, e_3, e_4, e_5, e_6\}$. Disto, temos:

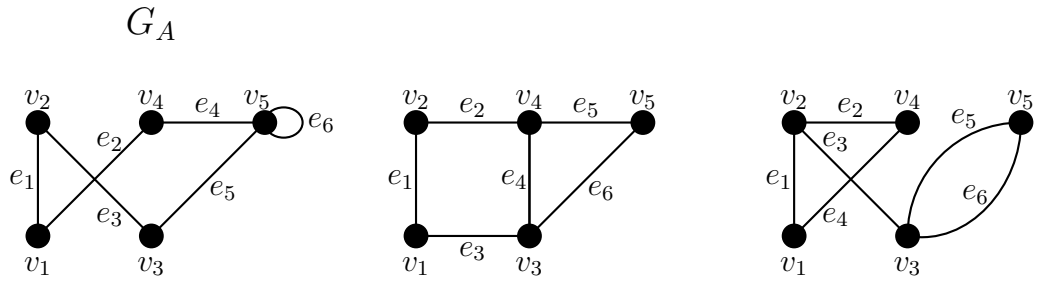


Figura 42 – Representação de grafos que possuem a mesma quantidade de vértices e arestas, com relações distintas.

Observe que, com o mesmo conjunto de arestas e vértices, podemos construir grafos distintos, mudando apenas as relações entre as arestas. Dessa forma, a aresta é o elo entre dois vértices, tornam-os um adjacente ao outro. Com isso, sendo $u, v \in V(G)$, quando u e v são extremidades da aresta $e \in E(G)$, então podemos representar como uv, em vez de e e, assim, torna a aresta vai representativa, indicando explicitamente suas extremidades.

9.1 Conceitos e definições

Iremos apresentar, de uma forma simplificada, alguns conceitos atribuído aos grafos. Com essas definições, podemos agrupar os grafos que possua característica em comum. Além disso, tornamos nosso estudo mais organizado e fluido.

9.1.1 Vértices adjacentes e isolados, adjacência e grau de um vértice

Dois vértices são ditos adjacentes, ou vizinhos, quando são conectados entre si por uma aresta. Em outras palavras, $u \leftrightarrow v$, se, e somente se, $uv \in E(G)$ (utiliza-se "$\leftrightarrow$" para representa a relação de adjacência). Por exemplo, na Figura 42, no grafo G_A, o vértice v_1 é adjacente ao vértice v_2, e vice-versa, uma vez que ambos são as extremidades da aresta e_1 e, com isso, $v_1 \leftrightarrow v_2$.

A vizinhança de um vértice v, escrevemos $N(v)$, é o conjunto de vértices vizinhos à v. Na Figura 42, $N_{G_A}(v_1) = \{v_2, v_4\}$. Um vértice é dito isolado quando sua vizinhança é vazia, ou melhor, no caso de não possui nenhuma relação com outro vértice.

O grau do vértice $deg(v)$ é a quantidade de arestas que incidem no vértice. Para a Figura 42, $deg_{G_A}(v_1) = 2$, $deg_{G_B}(v_2) = 3$. Podemos dizer que o grau de um vértice é o tamanho de sua vizinhança. Definimos o grau máximo de um grafo G como $\Delta(G)$ e o mínimo $\delta(G)$.

9.1.2 *Loop*, múltiplas arestas e grafos simples

Um *loop* representa uma aresta onde os vértices da extremidade é o mesmo, de forma que um vértice é vizinho de si mesmo. Da Figura 42, a aresta e_6, do grafo G_A é um *loop*.

Múltiplas arestas, ou arestas paralelas, são arestas que possuem os mesmas extremidades. Temos que, em G_C, as arestas e_5 e e_6 são paralelas. Um grafo é dito simples, quando não possui múltiplas arestas. Os grafos G_A e G_B são simples.

9.1.3 Subgrafos

Um grafo H é dito subgrafo de G, denotado por $H \subseteq G$, se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$[9]. Em outras palavras, H é subgrafo de G quando é obtido por meio de remoções de vértices ou arestas de G. Dessa forma, se para toda aresta $uv \in E(H)$, então $uv \in E(G)$. Um subgrafo obtido a partir da remoção de vértices é dito subgrafo induzido. Com isso, a remoção de um vértice ocasiona na remoção de todos as arestas que o coincide para temos um subgrafo induzido.

9.1.4 Grafos completos e vazios

Um grafo G é dito completo quando todos os seus vértices são vizinhos entre si, ou seja, $N_G(v) = V(G) - \{v\}$, para todo vértice $v \in V(G)$. Representamos um grafo completo como K_i, onde i é a quantidade de vértices no grafo.

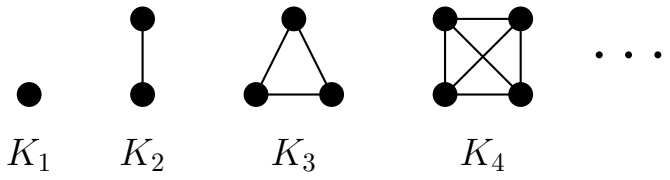


Figura 43 – Representação de grafos completos e simples.

Um grafo G é vazio se $E(G) = \emptyset$. Em outras palavras, qualquer grafo que não possua aresta é vazio. Com isso, para todo vértice $v \in V(G)$, $deg_G(v) = 0$.

9.1.5 Grafos complementares

O complementar $\overline{G}$ de um grafo simples G é o grafo simples com o conjunto de vértices $V(G)$, definido por $uv \in E(\overline{G})$ se, e somente se $uv \notin E(G)$[9]. O grafo $\overline{G}$ possui a mesma quantidade de vértices que G, porém, onde houver arestas em G, não há em $\overline{G}$. Dessa forma, podemos nos referir à $\overline{G}$ como o inverso de G.

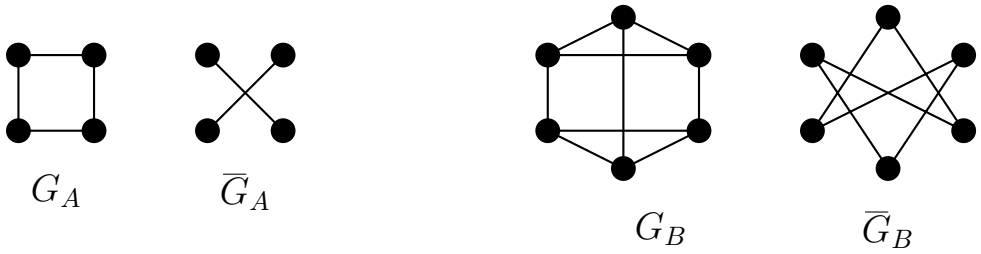


Figura 44 – Exemplos de grafos juntamente com seu complementar.

9.1.6 Cliques, conjunto independente e grafos bipartidos

Um clique em um grafo G é conjunto de pares de vértices adjacentes. De outra forma, temos que um clique $C \subseteq V(G)$, tal que para todo $u, v \in V(C)$, $uv \in E(G)$. Dessa forma, podemos dizer que um clique é um subgrafo induzido de G completo, isso porque, se um conjunto de vértices é adjacente para cada par de vértices, então tal conjunto é completo. O clique máximo é o clique como maior número de vértice. Nomeamos $\omega(G)$ o número de vértices de um clique máximo.

Ao contrário da clique, um conjunto independente é o conjunto de vértices que não são vizinhos entre si. Ou seja, o conjunto independente $I \subseteq V(G)$, tal que, para todo $u, v \in v(I)$, $uv \notin E(G)$. O número máximo de vértice em um conjunto independente é nomeado $\alpha(G)$.

Um grafo é dito bipartido se $V(G)$ é o resultado da união de dois conjuntos independentes disjuntos. Com isso, se conseguirmos dividir um grafo em duas partes, onde os elementos do conjunto não são vizinhos entre si, então esse grafo é bipartido. Esses conjuntos é chamados de partições. Tal propriedade pode ser útil, por exemplo, para realizar uma distribuição de m funções para n pessoas.

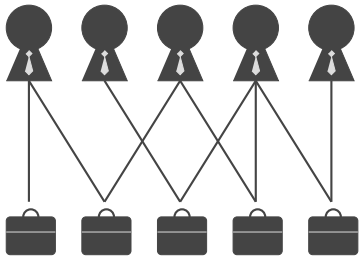


Figura 45 – Representação de uma distribuição de tarefas em uma empresa. Fonte: Autor.

Um grafo bipartido G, onde $G = P_1 \cup P_2$ (sendo P_1 e P_2 as partições de G), é completo se para todo $v \in V(P_1)$, $N_{P_1}(v) = V(P_2)$. Vale salientar que existe grafos com mais de duas partições.

9.1.7 Passeio, trilhas, caminhos e ciclos

Um passeio em um grafo G é uma sequência $v_1, e_1, v_2, e_2, v_3, e_3, ..., e_k, v_k$ de vértices e arestas, de modo que os vértices v_{i-1} e v_i são extremidades da aresta e_i, onde $1 \leq i \leq k$[9]. Uma trilha em G é o um passeio em que todas as arestas são distintas. Um caminho em G é um passeio em que todos os vértices são distintos.

Podemos relacionar estas definições com suas nomenclaturas. Quando realizamos um passeio em uma floresta, normalmente passamos pelo mesmo lugar, seja para tirar uma foto, ou mesmo para vê algo que passou despercebido na primeira vez. Ao realizar uma trilha, seguimos uma rota que passa por diversos pontos, podendo repeti-las em caminhos distintos, uma vez que o desejado é fazer diferentes trajetos. Por fim, para percorrer um caminho, não faz sentido repetir lugares ou trajetos, de forma que desejamos apenas chegar ao ponto final.

Um passeio ou trilha é dito fechado caso o vértice inicial e o final seja o mesmo, (também pode ser nomeado circuito). Um caminho que possua essas características é denominado ciclo ou caminho fechado (isso pode parecer estranho, uma vez que o último vértice coincide com o primeiro, mas vamos desconsiderar isso). Geralmente, usamos para representá-lo C_i, onde i é o número de vértices do grafo. Um grafo sem ciclos e dito acíclico.

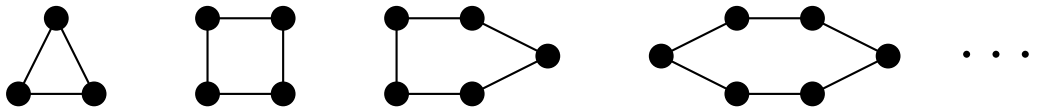


Figura 46 – Representação de ciclos.

9.1.8 Conectividade em grafos

Um grafo G é dito conexo se existe um caminho para todo par de vértice de G. Se existe um caminho entre os vértices v e u, então v é conectado à u. Caso não seja conexo, o grafo é dito desconexo.

Os componentes de um grafo G são os subgrafos maximais conexos[9] (ou seja, não podemos adicionar nenhum outro vértice ou aresta no subgrafo). Um componente é dito trivial se não possui arestas. Caso um componente seja trivial, ele necessariamente possui um vértice, sendo chamado de vértice isolado.

Uma aresta de corte, ou ponte, é uma aresta que, ao removê-la, aumenta o número de componentes. Semelhantemente, um vértice de corte é um vértice que aumenta o número de componentes ao deletá-lo. Dessa forma, quando escrevemos $E(G) - e$ ou

$V(G) - v$, estamos representando um subgrafo obtido da remoção de uma aresta ou vértice.

Um conjunto de separação de um grafo G é um conjunto de vértices $S \subseteq V(G)$, tal que $G - S$ aumenta o número de componentes. A conectividade de G, nomeamos $\kappa(G)$, é o número mínimo de vértices necessário para aumentar a quantidade de componentes. Um grafo G é dito k-conexo, ou k-conectado, se ele possui conectividade pelo menos k.

Um conjunto de corte de um grafo G é o conjunto de arestas $F \subseteq E(G)$, tal que $G-F$ aumenta o número de componentes. A aresta-conectividade de G, nomeamos $\kappa'(G)$, é o número mínimo de arestas para desconectar o grafo. Um grafo é k-aresta-conectado se ele possui aresta-conectividade pelo menos k.

9.1.9 Isomorfismo

Um isomorfismo de um grafo simples G para um grafo simples H é uma bijeção $f : V(G) \to V(H)$, tal que $uv \in E(G)$ se, e somente se, $f(u)f(v) \in E(H)$. Se existe isomorfismo entre G e H, dizemos que G é isomórfico a H, e representamos $G \cong H$, ou ainda, G e H são isomorfos. Em outra palavras, "dois grafos são isomorfos se é possível alterar os nomes dos vértices de um deles de tal modo que os dois grafos fiquem iguais"[10].

9.1.10 Árvores em Grafos

A árvore é um grafo conexo acíclico e, além disso, simples uma vez que *loops* e múltiplas arestas formam ciclos. Uma floresta é um grafo composto por um ou mais subgrafos que são árvores. A folha de uma árvore é todo vértice com grau 1.

Um subgrafo gerador do grafo G é um subgrafo $T \subseteq G$, tal que $V(G) = V(T)$. Uma árvore geradora é um subgrafo gerador que é uma árvore[9].

9.1.11 Emparelhamento e cobertura

Um emparelhamento em um grafo G é um conjunto M de arestas que não compartilham vértices como extremidades, e que não são *loops*. Em outras palavras, todo vértice de G incide em no máximo um elemento de M. O emparelhamento é comparado ao conjunto independente, pois ambos cria um conjunto de elementos isolados entre si. O tamanho máximo de uma emparelhamento é nomeado $\alpha'(G)$.

Uma cobertura de arestas de um grafo G é o conjunto L de arestas tal que todo vértice de G incide em alguma aresta de L. Uma cobertura de vértice L' de G refere-se ao conjunto de vértices em que toda aresta incide em algum vértice de L'. O número máximo de cobertura de vértices é definido por $\beta(G)$ e de aresta por $\beta'(G)$.

9.1.12 Coloração de Grafos

Uma k-coloração de um grafo G é uma rotulagem $f : V(G) \to S$, onde $|S| = k$. Os rótulos são as cores; os vértices de uma mesma cor são chamadas de classe de cor. Um k-coloração é dita própria se os vértices adjacentes possui cores distintas. Um grafo é dito k-colorível se ele possui um k-coloração própria[9].

O número cromático $\chi(G)$ é o menor número de cores necessárias para colorir G. Um grafo é k-cromático se $\chi(G) = k$. Se $\chi(H) < \chi(G) = k$ para todo subgrafo próprio $H \subseteq G$, então G é k-crítico.

9.1.13 Grafos planares

Um grafo G é dito planar se ele pode ser desenhado em um plano (como uma folha de papel), de forma que não exista cruzamento de arestas. Um desenho no plano é chamado imersão plana, ou mapa, de G. Um grafo plano é um grafo planar que está imerso no plano.

O grafo planar divide o plano em um determinado número de regiões, denominada faces, incluindo regiões totalmente limitadas por arestas e uma região exterior ilimitada, chamada de face externa[11]. As arestas que delimitam uma face é dita fronteiras de face.

9.2 Digrafos

Um grafo orientado D, ou digrafo, é uma tripla ordenada que consiste em um conjunto de vértices $V(D)$, um conjunto de arcos $A(D)$, e uma relação que atribui aresta para um par ordenado de vértices.

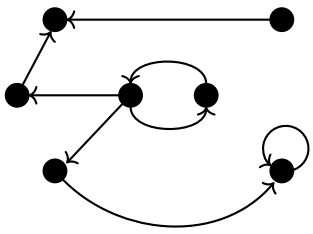


Figura 47 – Exemplo de um digrafo.

Assim como os grafos, podemos representar os arcos (que são as arestas no grafo) a partir de sua extremidade, ou seja, o arco $a = uv$, onde u e v são as extremidades de a. Observe que, para digrafos, $uv \neq vu$. O vértice u é dito cauda de a e o vértice v é dito cabeça de a. Dessa forma, dizemos que a sai de u e aponta para v.

As propriedade definidas para grafos possuem alguma diferenças em relação aos digrafos. Por exemplo, quando consideramos o grau de um vértice v, existe o conceito de

grau de entrada, denotado por $deg^-(v)$, que representa o número de arcos que apontam para v; e o de grau de saída, denotado $deg^+(v)$, que é o número de arcos que saem de v.

Um aplicação dos digrafos, bastante conhecida, é para representar um autômato finito, que é uma máquina de estados em que, dado uma *string* de entrada, é satisfeita ou não, dependendo da linguagem do autômato. Os vértices representam o estado e os arcos representam as transições, sendo rotulados com o caractere que precisa ser satisfeito naquela instância. Esse tópico é objeto de estudo de Linguagens Formais e Autômatos.

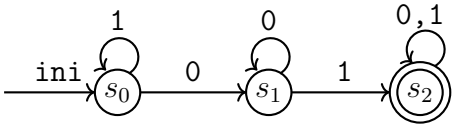


Figura 48 – O diagrama de transições que aceita todos os strings, onde $\sum = \{0, 1\}$, que contêm o substring 01.

9.3 Grafos como estrutura de dados

Para aplicar os conceitos apresentados de grafos para um computador, devemos definir como vamos representar cada vértice e aresta, de tal forma que mantenha os relacionamentos e propriedades. Existem diversas formas de representar um grafo, cada um com seus pontos positivos e negativos.

Em nossos exemplos, definimos os vértices representados por números inteiros. A representação das arestas dependerá do modelo que será implementado. Será apresentado apenas soluções para grafos não orientados.

9.3.1 Matriz de adjacência

A matriz de adjacência de um grafo G é um matriz *booleana* M, tal que

$$M = \begin{bmatrix} x_{00} & x_{01} & \dots & x_{0n} \\ x_{10} & x_{11} & \dots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n0} & x_{n1} & \dots & x_{nn} \end{bmatrix}$$

com $n = |V| - 1$, sendo que, com $i, j \in V(G)$,

$$x_{ij} = \begin{cases} 1, & \text{se ij} \in E(G) \\ 0, & \text{caso contrário.} \end{cases}$$

Para implementarmos a matriz de adjacência, usaremos C. Ademais, veremos que nessa matriz os elementos da posição $x_ij = x_ji$. Isso ocorre pois lidamos com grafos não

orientados e, nesse caso, não importa a ordem dos vértices, porém para digrafos, eles teriam valores ocasionalmente distintos. Iremos armazenar em ambas posições equivalentes.

Inicialmente definimos a estrutura, que armazena, além da matriz, a quantidade de vértices do grafo.

```
1 struct grafo {
2    int nv;
3    int **matriz;
4 };
5
6 typedef struct grafo Grafo;
```

Agora, devemos inicializar a matriz. Para isso basta utilizar um laço e determinar o valor de todas as posições como 0, uma vez que estamos criando um grafo vazio, inicialmente.

```
1 void cria_grafo(Grafo* g, int n)  {
2    g->nv = n;
3    g->mat = malloc(n * sizeof (int));
4    int i;
5
6    for (i = 0; i < n; ++i)
7       g->mat[i] = calloc(n, sizeof (int));
8 }
```

Com a função *calloc*, simplificamos nosso código, uma vez que ele atribui valores padrão (no caso 0) para as linhas da matriz.

Para inserir uma aresta basta mudar o valor referente aos vértices que serão a extremidade da aresta para 1. A remoção é semelhante, mas mudamos a posição para o valor 0.

```
1 void inserir(Grafo* g, int u, int v) {
2    g->matriz[u][v] = g->matriz[v][u] = 1;
3 }
4 void remover(Grafo* g, int u, int v) {
5    g->matriz[u][v] = 0 = g->matriz[v][u] = 0;
6 }
```

Para percorrer a matriz, basta utilizar dois laços, iterando entre as linhas e colunas. O espaço necessário para armazenar uma matriz de adjacência é proporcional ao número de vértices ao quadrado, ou seja, $|V|^2$.

9.3.2 Lista de adjacência

A lista de adjacência é semelhante à matriz, porém utiliza uma lista encadeada no lugar de um dos *arrays*. Ou seja, utilizamos um array para armazenar os vértices e uma lista encadeada para armazenar as relações que representam as arestas. Dessa forma, temos uma economia de espaço, porém perdemos o acesso em tempo constante para todas as posições.

Para implementá-lo, definimos a estrutura que armazena o número de vértices e uma lista encadeada. Para a lista, usaremos as operações desenvolvidas na seção 3, porém os elementos serão inteiros em vez de caracteres.

```
1 struct grafo {
2    int nv;
3    Celula *lista;
4 };
5 typedef struct grafo Grafo;
```

Para criar o grafo devemos apenas alocar memória para a estrutura do grafo, com as listas inicialmente nulas, criando assim um grafo vazio.

```
1 void cria_grafo(Grafo* g, int n)  {
2    g->lista = malloc(n*sizeof(Celula));
3    g->nv = n;
4    for (int i = 0; i < n; i++)
5       g->lista[i] = NULL;
6 }
```

Na inserção de arestas, basta adicionarmos um elemento na lista. Mas, antes disso, é necessário verificar se o elemento já foi inserido anteriormente. Para tal, fazemos uma busca na lista e, caso encontre o elemento, não realizamos a inserção.

```
1 void inserir_grafo(Grafo* g, int u, int v) {
2    if (buscar(g->lista[u], v) != NULL) return;
3
4    inserir(&(g->lista[u]), v);
5    inserir(&(g->lista[v]), u);
6 }
```

Na remoção de uma aresta, devemos deletar os elementos nas listas de cada vértice.

```
1 void remover_grafo(Grafo* g, int u, int v) {
2    removerNo(&(g->lista[u]), v);
3    removerNo(&(g->lista[v]), u);
4 }
```

Por fim, podemos imprimir para testes da seguinte forma:

```c
void imprimir(Grafo *g, int n) {
   int i;
   for(i = 0; i < n; i++) {
      printf("%i -> ", i);
      imprimirNo(g->lista[i]);
      printf("\n");
   }
}
```

Exercício 54. *Determine se as afirmações abaixo são verdadeiras ou falsas. Justifique.*

(a) *Todo grafo desconexo tem um vértice isolado.*

(b) *Um grafo é conexo se, e somente se, existe algum vértice que é conectado a todos os outros vértices.*

(c) *Se um trilha maximal em um grafo não é fechada, então as suas extremidades têm grau ímpar.*

Exercício 55. *Prove que todo grafo com n vértices e com n arestas contém um ciclo.*

Exercício 56. *Usando grafos completos e argumentos de contagem (sem usar álgebra), prove que:*

$$\binom{n}{2} = \binom{k}{2} + k(n-k) + \binom{n-k}{2}$$

Exercício 57. *Seja G um grafo simples com n vértices diferente de K_n. Prove que se G não é k-conexo, então G tem um conjunto de separação de tamanho $k-1$.*

Exercício 58. *Prove que $\kappa(G) = \kappa'(G)$ quando G é um grafo simples com $\Delta(G) \leq 3$*

Exercício 59. *Prove ou refute: Se $G = F \cup H, então o \chi(G) \leq \chi(F) + \chi(H)$.*

Exercício 60. *Prove ou refute: Para todo grafo G, $\chi(G) \leq n(G) - \alpha(G) + 1$.*

Exercício 61. *Prove que todo grafo planar simples tem um vértice de grau no máximo 5.*

Exercício 62. *Desenvolva um algoritmo para decidir se um vértice v é isolado.*

Exercício 63. *Desenvolva uma função para calcular o grau de um vértice v.*

Exercício 64. *Escreva uma função que construa um grafo completo com n vértices.*

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>.

5 GOODRICH, M. T.; TAMASSIA, R. *Estruturas de Dados & Algoritmos em Java*. [S.l.]: Bookman Editora, 2013.

6 SZWARCFITER, J. L.; MARKENZON, L. *Estruturas de Dados e seus Algoritmos*. [S.l.]: Livros Tecnicos e Cientificos, 1994. v. 2.

7 TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. *Estruturas de dados usando C*. [S.l.]: Pearson Makron Books, 2004.

8 SOUZA, J. E.; RICARTE, J. V. G.; LIMA, N. C. de A. Algoritmos de ordenação: Um estudo comparativo. *Anais do Encontro de Computação do Oeste Potiguar ECOP/UFERSA (ISSN 2526-7574)*, n. 1, 2017.

9 WEST, D. B. et al. Introduction to graph theory. Prentice hall Upper Saddle River, NJ, v. 2, 1996.

10 FEOFILOFF, P.; KOHAYAKAWA, Y.; WAKABAYASHI, Y. Uma introdução sucinta à teoria dos grafos. 2011.

11 CAVALCANTE, F. N. S.; SILVA, S. D. S. Grafos e suas aplicações. *São Paulo*, 2009.