**PROGRAMAÇÃO III**

AULA 1

Prof. Vinicius Pozzobon Borin

**CONVERSA INICIAL**

O que são estruturas de dados?

Na programação, estruturas de dados são maneiras de organizar e colecionar dados. E a forma como os dados ficam organizados dentro da memória, o acesso a eles e suas manipulações caracterizam esse estudo.

Você já estudou programação antes e, certamente, já manipulou algumas estruturas de dados, independentemente da linguagem. Se você já trabalhou com linguagem Python, deve conhecer as estruturas de listas, tuplas e dicionários. No Java, e/ou no C/C++, você deve se lembrar do vetor (*array*), *string*, *struct*, *map*, entre outros. Até então, você sabia utilizar essas estruturas de dados. Neste estudo, você irá aprender a construir tais estruturas do zero – assim como outras –, conhecendo suas caracteristicas, aplicações, vantagens e desvantagens.

Vamos nos aprofundar em programação conhecendo estruturas de dados, estudando, majoritariamente, códigos sendo escritos em linguagem Python.

**TEMA 1 – PESQUISA EM UM CONJUNTO DE DADOS**

Todo e qualquer problema computacional que possa ser solucionado por meio de um algoritmo apresenta inúmeras soluções distintas. Cada desenvolvedor é capaz de pensar em uma solução ligeiramente diferente da de outro.

O fato de termos diferentes algoritmos possíveis para solucionar um problema nos traz uma pergunta pertinente: Todos os algoritmos terão o mesmo desempenho ao executar? Existem algoritmos mais – ou menos – eficientes que outros? Ademais, quais métricas são empregadas para analisar e comparar os algoritmos?

Vamos tentar responder a todas essas perguntas ao longo dessa etapa. Iniciaremos nosso estudo comparando dois algoritmos distintos para resolver o mesmo problema: **a busca dentro de um conjunto de dados**.

**1.1 PESQUISA SEQUENCIAL**

Vamos supor que você e um amigo estão jogando um jogo de adivinhação de números, no qual seu amigo gostaria que você adivinhasse o número que está pensando dentro de um intervalo de 1 a 100 valores. A cada tentativa sua, seu amigo irá indicar se o valor está baixo ou alto em relação ao que ele pensou.

Imagine que o valor pensado por seu amigo é o 99, e você decidiu tentar adivinhar os valores assim: 1, 2, 3, 4, 5 etc. – até atingir o valor correto. A cada tentativa, seu amigo informa que o valor está baixo, e você chuta o próximo valor. Observe a Figura 1.

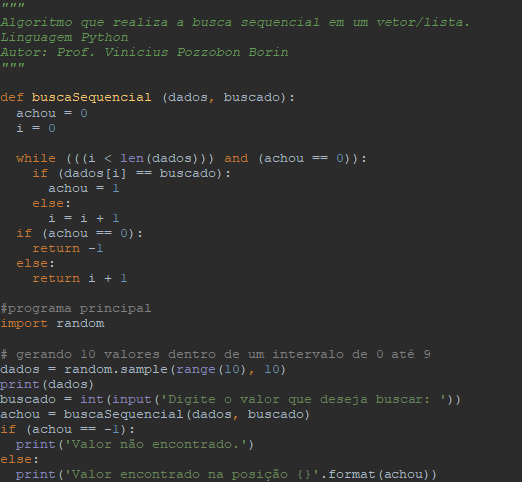
Figura 1 – Pesquisa sequencial: sequência de tentativas – em laranja, as tentativas; em verde, o acerto



Ao fazer esse tipo de tentativa, você está eliminando da lista apenas um valor por vez. Como o valor correto é 99, você precisa de 99 chances para acertar.

A implementação dessa lógica em um algoritmo nos remete a um simples laço de repetição. Se estivermos manipulando uma estrutura de dados de vetor, faremos uma varredura desse vetor iniciando no índice zero, até o final dele.

A seguir, o código apresenta a função que recebe como parâmetro um vetor e o valor a ser buscado, retornando, como saída, a posição no vetor onde o valor está.



**1.2 PESQUISA BINÁRIA**

Achou a busca sequencial ineficiente? Será que não conseguiríamos trabalhar com uma lógica mais eficiente para resolver o problema da busca? Certamente.

Agora, imaginemos que você irá tentar iniciar a adivinhar o número pelo valor 50. Seu amigo irá responder: “Está baixo!”. Sabendo disso, você acabou de descobrir que o valor a ser adivinhado está entre 50 e 100, ou seja, acabou de eliminar metade dos números em uma só tentativa.

Na próxima tentativa, se chutar o valor 75, estará novamente quebrando o conjunto de valores ao meio. Se continuar dessa maneira, eliminando metade dos dados, estará sempre reduzindo seu conjunto de dados ao meio, até restar somente o valor buscado. Esse princípio de raciocino lógico é chamado de **dividir para conquistar**.

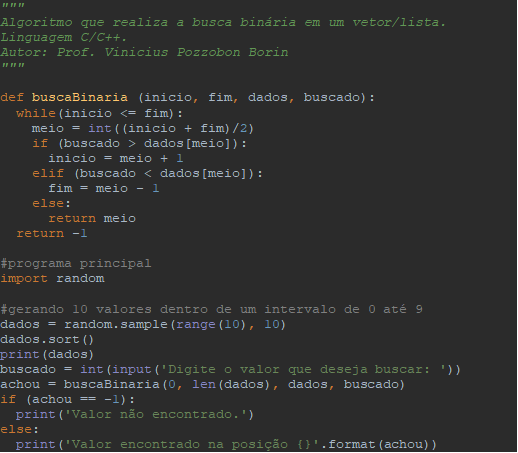
A Tabela 1 mostra todas as sete tentativas necessárias para localizar o dado 99 dentro do conjunto.

Tabela 1 – Pesquisa binária: total de tentativas e intervalos

|  |  |  |  |
| --- | --- | --- | --- |
| **Tentativa** | **Intervalo inicial** | **Intervalo final** | **Valor do chute** |
| **1** | 1 | 100 | 50 |
| **2** | 50 | 100 | 75 |
| **3** | 75 | 100 | 87 |
| **4** | 87 | 100 | 93 |
| **5** | 93 | 100 | 96 |
| **6** | 96 | 100 | 98 |
| **7** | 98 | 100 | 99 |

A implementação dessa lógica em um algoritmo será estudada logo a seguir. A função recebe como parâmetro o vetor a ser buscado, o dado a ser localizado e o intervalo dentro do vetor a ser buscado.

No algoritmo, note que ele inicia localizando o valor central do conjunto de dados e verificando se ele é o valor a ser buscado (testes condicionais). A alterações das delimitações dos intervalos ocorrem quando uma condicional resulta em verdadeiro.



Existe um revés, porém, na busca binária. O fato de o conjunto de dados precisar, obrigatoriamente, estar ordenado (note o método *sort* invocado no algoritmo principal). Caso contrário, será impossível saber se o valor procurado está na primeira, ou na segunda parte do conjunto de dados.

**1.3 A IMPORTÂNCIA DO CONJUNTO DE DADOS**

Talvez você esteja se perguntando: “E se o número a ser adivinhado fosse 1? A busca sequencial não encontrará a resposta na primeira tentativa, sendo melhor que a binária?”. De fato, será. Isso significa que o algoritmo é mais eficiente? Não, pois a ideia de eliminar um só número por vez permanece.

A maneira como os dados estão organizados dentro do conjunto de dados é de suma importância para o desempenho real do algoritmo. Caso o número a ser adivinhado seja 1, isso significa que as condições para o algoritmo sequencial funcionar bem estavam ótimas. Mas isso não significa que, ao compararmos a complexidade de ambos os algoritmos, a busca sequencial é melhor. Trataremos mais desse assunto ao longo dessa etapa, e vamos estudar como comparar matematicamente ambos os algoritmos, com o objetivo de definir qual tem melhor desempenho.

**TEMA 2 – ANÁLISE DE ALGORITMOS**

Vamos agora tentar, efetivamente, responder à seguinte questão: Como podemos comparar o desempenho de diferentes algoritmos para uma mesma aplicação? Podemos mensurar isso?

Quando queremos descobrir qual algoritmo é **mais eficiente** para resolver um problema, estamos falando do algoritmo de **menor complexidade**. O objetivo da verificação da complexidade é identificar como o desempenho do algoritmo cresce à medida que o tamanho do conjunto dos dados de entrada (*n*) cresce também. Ou seja, o algoritmo que consumir menos recursos, será de menor complexidade e mais eficiente. E o que é a complexidade do algoritmo?

Temos, portanto, dois tipos de complexidade de algoritmos:

* **Complexidade de tempo**: é o tempo que um algoritmo leva para completar sua execução. A quantidade de instruções do código impacta diretamente em seu desempenho.
* **Complexidade de espaço**: é a quantidade de memória requerida para a execução de um algoritmo. A quantidade de variáveis e seus tamanhos impactam diretamente em seu desempenho.

Muitas vezes, um algoritmo pode ser somente complexo em tempo, mas não em espaço, e vice-versa. De todo modo, nesse material, focaremos nossos estudos somente na complexidade de tempo dos algoritmos, deixando a complexidade de espaço de lado.

Para um mesmo algoritmo, o tempo levado para ele executar depende, majoritariamente, de três fatores:

1. **Tamanho do conjunto de dados de entrada (n)**: quanto mais dados temos para manipular, mais tempo.
2. **Disposição dos dados dentro do conjunto**: a ordem com que os dados estão organizados no conjunto implicada em distintas situações.
3. **Quantidade de instruções a serem executadas**:entendemos por instrução cada linha de código de programação de alto nível, independentemente do *hardware*.

Note que os dois primeiros itens independem do algoritmo escolhido para resolver o problema. Porém, a quantidade de instruções é diretamente proporcional à eficiência do algoritmo construído.

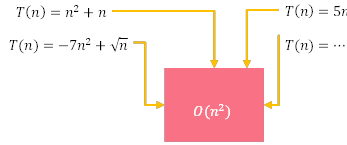
**2.1 NOTAÇÃO *BIG-O***

Para podermos comparar a complexidade dos algoritmos, podemos analisá-los matematicamente. A notação mais comum adotada na literatura para comparar algoritmos e dizer o quão rápido um algoritmo é, é a **notação *Big-O***(ou “Grande-O”). Para compararmos a complexidade de algoritmos, devemos encontrar a função matemática que descreve cada um, e compará-las.

Assumindo que um conjunto T(n) de funções é tido como todas as funções que contêm ordem de grandeza menor ou igual a G(n), quando falamos de notação *Big-O*, dizemos que é um conjunto de funções T(n), que pertencem à ordem de grandeza G(n).

Por exemplo: se , isso significa que a ordem de grandeza de todas as funções que pertencem a esse conjunto é . A seguir, podemos ver diferentes funções que pertencem a esse conjunto.

Figura 2 – Conjunto de funções T(n) que pertencem à ordem de grandeza de



Podemos dizer que cada equação T(n) apresentada na Figura 2 é um algoritmo distinto para resolver um problema. Todos esses algoritmos apresentam a mesma ordem de grandeza e, portanto, a mesma **complexidade assintótica** (notação *Big-O*).

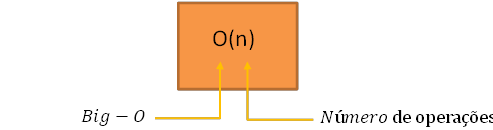
**Desse modo, para sabermos a complexidade de um algoritmo utilizando a notação *Big-O*, basta encontrarmos o termo de maior grau da equação que o descreve.**

A notação *Big-O* também é tida como o**limite inferior (pior caso)** de um algoritmo. Ou seja, o pior cenário que o algoritmo pode enfrentar para executar. Desse modo, com o *Big-O* teremos um parâmetro que indica que o algoritmo nunca será pior do que aquele cenário.

Há outras notações que podem ser investigadas. A que indica o **limite superior** **(melhor caso – Big-Ω)**,e o **caso médio (Big-θ)**. Essas notações não serão trabalhadas nesse estudo.

A notação *Big-O* é sempre representada pela letra “O” maiúscula. Dentro dos parênteses, temos o termo de maior grau da equação, o qual aprenderemos a calcular ainda nessa etapa. Note, na Figura 3, como se dá essa representação.

Figura 3 – Representação da notação *Big-O*



O termo de maior grau colocado na notação não representa diretamente a equação do tempo que o algoritmo irá levar para executar, pois o tempo irá depender do *hardware*, e estamos tentando abstraí-lo nessa análise.

Apesar disso, ainda podemos comparar tempos se fizermos algumas suposições. Imaginemos que temos um algoritmo que realiza a soma de *10*valores de um vetor (*n = 10*). Assumindo que cada somatório leva 1 ms para acontecer, quanto tempo O(n) e O(n²) levarão para executar essa soma?


O O(n²) levará bem mais tempo para concluir o cálculo. Portanto, ele é mais complexo em tempo de execução, ou seja, menos eficiente.

**TEMA 3 – ENCONTRANDO O *BIG-O* DE ALGORITMOS**

Como efetivamente encontramos os *Big-O* dos algoritmos? Precisamos agora estudar e analisar diferentes situações de algoritmos para descobrir.

Podemos iniciar imaginando um programa sem laços de repetição, nem funções recursivas. Considere um programa qualquer que somente some dois valores simples. Esse tipo de algoritmo irá executar o código uma única vez e se encerrar. Dizemos que um programa assim não tem dependência do conjunto de dados de entrada, desse modo, seu *Big-O* é dito como constante: **O(1)**.

**3.1 LAÇO SIMPLES**

Vamos agora investigar um algoritmo com um laço de repetição simples. O código a seguir faz a varredura de um vetor e imprime, na tela, um valor. O que está sendo executado dentro do laço é irrelevante para nossa análise.

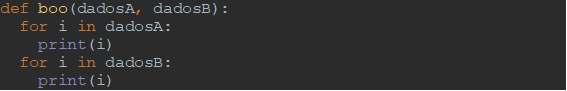
É importante compreender o que acontece quando um *loop* é executado. A cada iteração de um laço, todas as instruções dentro de sua estrutura serão executadas *n*vezes. Neste exemplo, *n,*é o tamanho de nosso vetor. Se o vetor/lista for de dimensão 10, o laço ocorre 10 vezes.



Na análise *Big-O*, sabemos que a expressão matemática que define o algoritmo representa a quantidade de instruções a serem executadas. A complexidade assintótica desse algoritmo será, portanto, **O(n).**

**3.2 PROPRIEDADE DA ADIÇÃO**

Vamos agora analisar uma propriedade matemática da notação *Big-O*. Sempre que houver um trecho de código seguido por outro que é independente, somamos suas notações *Big-O*. A seguir, veja dois laços de repetição simples, sendo executados um após o outro.



Em ações consecutivas, fazemos adição:

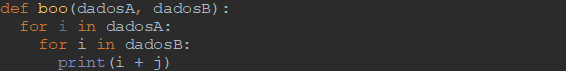


No *Big-O*, só nos interessa o termo de maior grau da equação. Portanto, negligenciamos o multiplicador dois na complexidade assintótica, ficando:



**3.3 PROPRIEDADE DA MULTIPLICAÇÃO**

Agora, sempre que houver um trecho de código aninhado a outro, ou seja, com relação de dependência entre eles, multiplicaremos suas notações. A seguir, observe dois laços de repetição simples sendo executados aninhados.

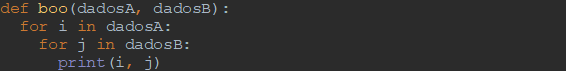


Em ações aninhadas, fazemos multiplicação:



**3.4 PROGRESSÃO ARITMÉTICA (PA)**

Vamos agora investigar um algoritmo com dois laços de repetição. Nesse caso, a análise é um pouco mais complexa. A seguir temos uma função na qual um laço está aninhado em outro. As instruções dentro dos laços são irrelevantes ao problema.



Quando temos um aninhamento de laços, precisamos analisar a quantidade de iterações geradas com base na dependência entre os laços. A maneira como essa dependência existe irá representar a quantidade de vezes que o laço interno irá executar, caracterizando uma progressão matemática.

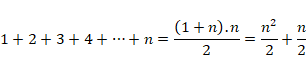
O caso mais comum é o de uma **série aritmética**, caracterizada como **uma sequência de números na qual a diferença entre dois termos consecutivos é constante**. Exemplos:

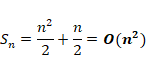

A soma dos *n*termos da progressão aritmética é dado por:



Em que é o valor do somatório no enésimo termo, é o enésimo termo da série, é o primeiro termo da série. Observe esse exemplo:



Para encontrarmos o *Big-O* dessa PA, precisaremos extrair da equação somente o termo de maior grau de crescimento. Para descobrirmos isso, podemos assumir que o *n*tende ao infinito, e substituir na equação, obtendo o resultado. Nesse caso, podemos negligenciar da equação o divisor 2, e também o termo de primeiro grau, ficando somente com .



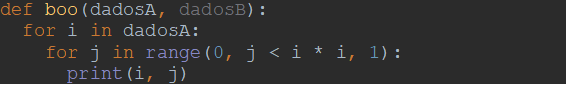
Um algoritmo com complexidade O(n²) normalmente terá dois laços de repetição aninhados. O laço da linha 4 depende do laço da linha 3. Se o tamanho do vetor for igual a 10, o segundo laço irá acontecer sempre 10 vezes o tamanho do vetor. Analisando o laço interno, temos uma PA constante, na qual:



**3.5 PROGRESSÃO GEOMÉTRICA (PG)**

**Uma série geométrica é uma sequência de números na qual há uma razão entre um número e seu sucessor**. Um algoritmo caracterizado por uma PG também irá trabalhar com dois laços aninhados.

Observe o exemplo a seguir: há dois laços aninhados, portanto, teremos O(n²), certo? Errado; precisamos analisar a progressão dos dados. Nesse cenário, o laço interno (linha 4), contém uma condição de parada na qual *j*depende dei*(j* *< i²)*.



Nesse caso, o laço interno estará sempre aumentando a quantidade de vezes que irá executar, conforme representado na Tabela 2.

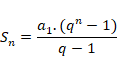
Tabela 2 – Variáveis dos laços que representam as iterações

|  |  |
| --- | --- |
| **I** | **J** |
| **0** | 0 |
| **1** | 1 |
| **2** | 4 |
| **3** | 9 |
| **4** | 16 |

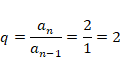
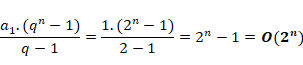
Vejamos outros exemplos de séries geométricas:


A soma dos *n*termos da progressão geométrica é dada por:



Em que é o valor do somatório no enésimo termo, é o primeiro termo da série e *q*é a razão da série (). Observe este exemplo:

**Assim, uma PG tem****, enquanto uma PA tem****.**

**TEMA 4 – DIVIDIR PARA CONQUISTAR**

Uma das maneiras mais clássicas de se pensar racionalmente na solução de um problema chama-se **princípio de dividir para conquistar**. Podemos criar algoritmos que funcionam com esse princípio.

O dividir para conquistar faz com que criemos uma estratégia na qual há um problema de dimensão *n*, e reduzimos esse problema em partes menores, até que ele vire a menor unidade possível daquele problema (problema-base).

Algoritmos recursivos trabalham bastante com tal princípio, mas nem sempre é necessário trabalhar com recursividade para adotar tal estratégia, que consiste em duas etapas:

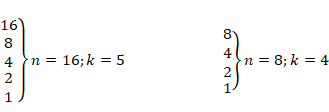
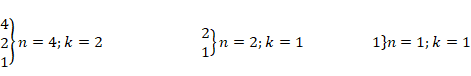
1. Descobrir o caso-base, que será sempre a menor parte possível para o problema;
2. Descobrir como reduzir o problema para que ele se torne o caso-base.

Vamos relembrar do algoritmo com que iniciamos essa etapa: o algoritmo de busca binária. Nele, pegamos um problema de dimensão *n* (o tamanho do nosso vetor é a dimensão), e dividimos o problema em partes menores, quebrando sempre ao meio, até obtermos a menor unidade possível dentro do vetor (caso-base). Essa menor unidade é a solução desse problema de busca.

Nem sempre localizar o caso-base irá finalizar nosso algoritmo. Iremos investigar, ao longo de nossos estudos, diferentes exemplos de dividir para conquistar, como algoritmos de ordenação de dados.

Como sabemos a complexidade de algoritmo que opera com esse princípio? Vamos, a seguir, realizar uma análise matemática utilizando o algoritmo de busca binária.

Assumindo que *n*é o conjunto de dados (tamanho do vetor de busca) e *k*é o número máximo de operações de busca possível, podemos ter:

Colocando isso em uma tabela, temos:

Tabela 4 – Conjunto de dados (*n*) e tentativas (*k*)

|  |  |  |
| --- | --- | --- |
| ***k*** | ***n*** | **N como potência de 2** |
| **5** | 16 |  |
| **4** | 8 |  |
| **3** | 4 |  |
| **2** | 2 |  |
| **1** | 1 |  |

Perceba que o número de tentativas sempre aparece no expoente quando colocado o conjunto de dados como uma base de 2. Portanto podemos generalizar o problema da seguinte maneira:


Para a notação *Big-O*, negligenciamos os termos de menor crescimento, ficando com somente:



Todo e qualquer algoritmo que opere com o princípio de dividir para conquistar terá uma complexidade logarítmica atrelada a ele.

**TEMA 5 – COMPLEXIDADE DA RECURSIVIDADE**

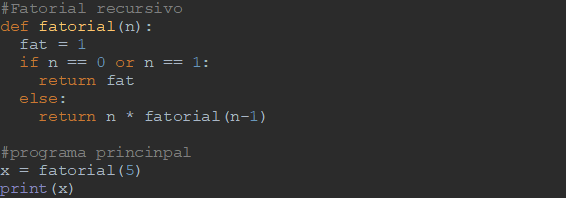
A complexidade de algoritmos recursivos demanda uma análise bastante minuciosa. De modo geral, para encontrarmos a complexidade de um algoritmo recursivo, devemos:

1. Calcular a complexidade de uma única chamada da função;
2. Expressar o número de chamadas recursivas por parâmetros de entrada;
3. Multiplicar o número de chamadas recursivas pela complexidade de uma chamada da recursão.

A fatorial é um caso clássico na literatura de recursividade, pois podemos escrever qualquer função recursiva, como nesse exemplo da fatorial de 5:



Note que a fatorial de 5 depende da fatorial de 4, que, por sua vez, depende da fatorial de 3, e assim por diante. Vejamos o caso do algoritmo de cálculo da fatorial utilizando recursividade.



Nele, há uma função chamada *fatorial*,que chama ela mesma até que a condição da linha 7 seja satisfeita. Vamos analisar as três etapas colocadas acima:

1. A complexidade de uma única chamada da função será O(1), uma vez que só temos instruções simples a serem executadas;
2. A quantidade de chamadas recursivas depende de quantas vezes a função é chamada novamente para cada instância da função aberta. No exemplo, cada função chama a si mesma uma só vez. Se o conjunto de dados for *n,* teremos *n*chamadas;
3. Assim, faremos: .

**5.1 EQUAÇÃO GERAL DA RECURSIVIDADE**

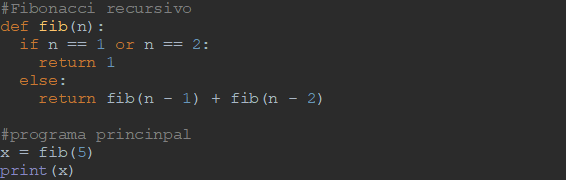
Existe uma equação que pode simplificar essa tarefa quando se trata de recursão. Porém, atente-se ao fato de que ela só é válida quando a quantidade de vezes que uma função recursiva invocar ela mesma duas ou mais vezes. A equação consiste em:



Restrições:

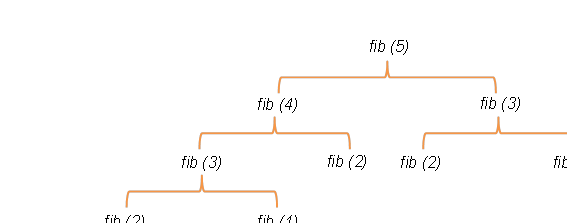
* O número de chamadas deve ser estritamente maior do que 1;
* Se o número de chamadas for igual a 1, teremos uma cadeia linear de chamadas (como na fatorial);
* Se a recursão ocorre em um *loop*, o número de chamadas depende de cada caso específico. Estudaremos isso melhor em breve.

Vejamos agora outro exemplo que envolve a série de Fibonacci implementada recursivamente.



Note que a função *fib,*ao ser chamada, invoca novamente ela mesma outras duas vezes (linha 6). Dessa maneira, uma só chamada da função *fib*irá resultar em 2n invocações da função *fib*. Isso nos remete a uma estrutura do tipo árvore binária, e o valor de *n*representa o número de níveis da árvore. Observe o exemplo de *fib(5)*:

Figura 4 – Invocações da função de Fibonacci em árvore binária



Para aplicarmos a equação geral da recursividade, precisamos, primeiro, descobrir a complexidade de uma só chamada recursiva. Será O(1), pois não temos instruções iterativas. A quantidade de nós da árvore será 2n, pois sempre iremos ramificar cada elemento em até dois outros. Assim, teremos:

O(uma chamada)=O(1)O(〖nº de chamadas〗^(nº níveis) )=O(2^n)

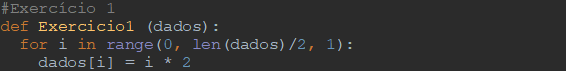
Portanto:



**5.2 EXERCÍCIOS**

Vamos encontrar a complexidade assintótica (*Big-O*) dos seguintes algoritmos escritos em linguagem Python.

**Exercício 1:**

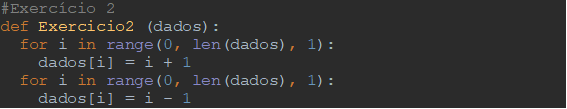


**Solução:**

Há um laço único de repetição. Para encontrarmos a complexidade *Big-O*, o que temos dentro do laço não nos interessa. O laço em si inicia em zero e vai até o tamanho da lista, dividido por dois. Ou seja, o *loop*é executado *n/2* vezes. Para o *Big-O*, negligenciamos o valor que está dividindo, assim teremos a complexidade de um laço simples:



**Exercício 2:**

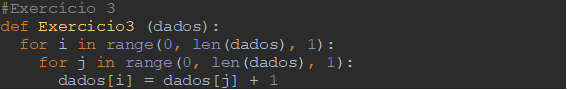


**Solução:**

Temos um laço de repetição e, após ele, outro. Assim, cada laço opera de maneira independente. Isso significa que cada laço representa O(n). Se somarmos ambas as complexidades, teremos O(2n), porém, para o *Big-O*, podemos negligenciar o multiplicador, ficando:



**Exercício 3:**



**Solução:**

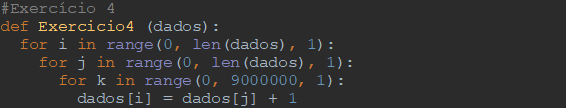
Temos um laço de repetição e, aninhado a ele, outro. Assim, existe dependência entre os laços. Nesse caso, precisamos verificar se as variáveis envolvidas no laço funcionam como uma PA, ou como uma PG.

O primeiro laço contém a variável *i*, que irá executar uma só vez até o tamanho da lista. Já o laço interno, que contém a variável *j*, irá executar o tamanho da lista para cada execução do laço externo. Se a lista tiver tamanho 10, o primeiro laço executará 10 vezes e, o segundo, 10 vezes 10.

O que mais nos importa aqui é que a cada nova execução do laço contendo a variável *j*, a quantidade de vezes será constante. Isso resultará em uma PA constante, portanto:



**Exercício 4:**



**Solução:**

Agora temos 3 laços aninhados. Todos eles executam suas variáveis do laço constantemente (PA). Sendo assim, poderíamos pensar que nossa complexidade seria, com 3 laços aninhados, , certo? Na verdade, não.

Note que no terceiro laço, com a variável *k*, a quantidade de iterações independe do conjunto de dados *n*, pois a iteração irá sempre ocorrer 9 milhões de vezes. Portanto:



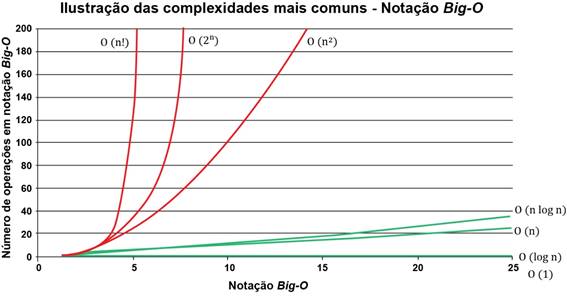
**FINALIZANDO**

Iniciamos esse estudo aprendendo que diferentes algoritmos para resolver o mesmo problema podem apresentar desempenhos distintos. Estudamos também como calcular o desempenho do algoritmo analisando sua complexidade de tempo de execução. Um algoritmo menos complexo é mais eficiente e, portanto, mais rápido.

Mensuramos a complexidade dos algoritmos pela complexidade assintótica do pior caso desse algoritmo (*Big-O*). Veja agora alguns tipos de algoritmos e suas respectivas complexidades:

* Algoritmo sem iterações nem recursão: O(1);
* Laço iterativo simples: O(n);
* Progressão aritmética (PA): O(n²);
* Progressão geométrica (PG): ;
* Dividir para conquistar: ;
* Recursão simples: O(n);
* Recursão em árvore binária: .

Figura 5 – Principais complexidades de algoritmos



**REFERÊNCIAS**

DROZDEK, A. **Estrutura de dados e algoritmos em C++**. Tradução da 4. ed. norte-americana. São Paulo: Cengage Learning, 2018.

KOFFMAN, E. B.; WOLFGANG, P. A. T. **Objetos, abstração, estrutura de dados e projeto usando C++**. Barueri: Grupo GEN, 2008

**PROGRAMAÇÃO III**

AULA 2

Prof. Vinicius Pozzobon Borin

**CONVERSA INICIAL**

Sabe quando você acessa um *site* de compras e seleciona para a página organizar os produtos por preço, do mais barato ao mais caro? Ou então você está trabalhando com uma planilha de nomes de pessoas no Excel e precisa que os dados fiquem ordenados alfabeticamente? Sempre por trás de tudo isso existe um algoritmo de ordenação de dados.

Uma das aplicações mais comuns na área da computação é a necessidade de ordenarmos um conjunto de dados. Dê uma busca rápida na internet pelo termo em inglês *sorting algorithm list* e verá, literalmente, dezenas de algoritmos diferentes para ordenar dados. Isso ocorre porque uma das categorias de algoritmos mais estudadas no meio científico sempre foram os algoritmos de ordenação de dados, e a comunidade científica vem desenvolvendo distintos algorítmos para esse fim.

Cada algoritmo de ordenação apresenta uma complexidade assintótica de tempo e de espaço próprias e, por consequência, aplicações específicas. Existem algoritmos de ordenação específicos para ordenação objetos em cenas de jogos de *videogames*, por exemplo.

Aqui, iremos focar nossos estudos em três algoritmos clássicos para ordenação de dados: ordenação por troca, também conhecida como ordenação bolha (*Bubble Sort*), ordenação por intercalação (*Merge Sort*) e ordenação rápida (*Quick Sort*). Analisaremos seu funcionamento e complexidade.

Os exemplos conduzidos nesta etapa tratarão do uso da linguagem Python, ordenando conjuntos de dados dentro de listas. Apesar dessa escolha, a lógica por trás desses algoritmos é análoga em todas as linguagens de programação e pode também ser aplicada a outros tipos de dados, como arrays e matrizes em C/C++ e Java.

**TEMA 1 – BUBBLE SORT**

O algoritmo do ***Bubble Sort***é também conhecido como **método da bolha,** ou **ordenação bolha**, ou ainda, **ordenação por troca**. Esse algoritmo é o de mais fácil entendimento e compreensão que temos quando nos referimos à ordenação de dados. Portanto, iniciamos por ele.

O *Bubble Sort*realiza a ordenação de um conjunto de dados por meio da comparação entre dois elementos adjacentes. O método faz uma varredura no conjunto de dados, do início ao fim em pares. Caso os números estejam em lugares invertidos, troca-se. Caso contrário, mantém-se onde estavam. Está muito confuso o entendimento? Não tem problema. Vamos verificar um exemplo mais completo a seguir.

**1.1 *BUBBLE SORT*: LÓGICA DE FUNCIONAMENTO**

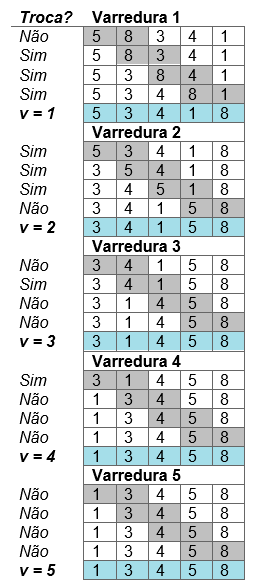
O algoritmo do *Bubble Sort*é clássico na literatura. Independentemente da linguagem a ser aplicada, sua lógica e funcionamento são imutáveis. Ela consiste em fazer varreduras no conjunto de dados, do início ao fim, pegando dois dados adjacentes e verificando se eles precisam ser trocados de ordem.

A quantidade de varreduras a serem feitas será igual ao tamanho do conjunto de dados. Exemplo, se tivermos um conjunto de dados com cinco dados, varreremos todo o conjunto cinco vezes.

Imagine agora que queremos ordenar o conjunto desornado de dados, contendo: , crescentemente. O algoritmo pegará os dois primeiros números (5 e 8) e irá compará-los para efetuar a troca. Uma vez comparado, avança-se para os números 8 e 3 e realiza-se o mesmo processo e, assim, indo até o final do conjunto de dados.

Na tabela a seguir, é apresentado todo o processo de ordenação, iteração por iteração, para as cinco varreduras. Em cinza, estão os dados comparados aos pares naquela iteração. Em azul, está o resultado final para aquela varredura. Vamos do início ao fim no conjunto de dados cinco vezes. Na coluna da esquerda, é apresentado se houve troca, ou não, naquela linha.

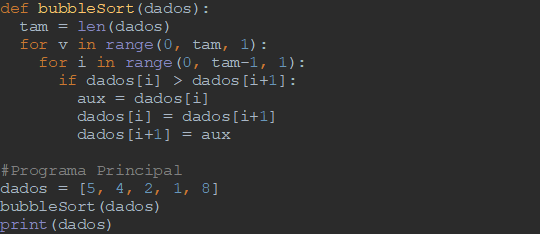
Tabela 1 – *Bubble Sort*: ordenando o conjunto de dados:.



Note que, na primeira e na segunda varredura, temos três trocas de dados. Na terceira e na quarta, somente uma. Na quinta varredura, nenhuma troca ocorre. O conjunto de dados já ficou ordenado ao final da iteração 4.

**1.2 *BUBBLE SORT*: ALGORITMO**

O *Bubble Sort* consiste em dois laços de repetição aninhados e uma condicional dentro deles. Veja:



O primeiro laço (variável *v*) é quem diz a quantidade de vezes em que a varredura deve ocorrer. Por esse motivo, fazemos um laço *for* iniciando em zero e indo até o tamanho do conjunto de dados, ou seja, de 0 até 5 no exemplo acima. Como na linguagem Python, o laço sempre se encerra no valor imediatamente anterior ao valor de parada, nosso laço irá ser realizado 5 vezes (de 0 até 4).

O segundo laço, aninhado ao primeiro, inicia 0 e vai até o tamanho dos dados menos um (*­tam - 1*). Mas por que essa condição de parada? Porque a última posição da lista é 4. Como as comparações são feitas aos pares, a última comparação feita deverá ser: *if dados[3] > dados[4]*. Ou seja, de maneira genérica, é equivalente a: *if dados[tam-2] > dados[tam-1]*.

Caso a condicional simples resulte em verdadeiro, significa que os dados estão no lugar errado e precisam ser trocados. Dentro dessa condicional, temos então a troca. Note a existência de uma variável auxiliar chamada *aux*. Essa variável serve para, temporariamente, auxiliar na troca dos valores.

**Saiba mais**

O algoritmo apresentado realiza a ordenação de forma crescente. Então talvez você esteja pensado agora: Como faço se quiser ordenar de maneira decrescente?

Para concretizar isso, basta que inverta a condição utilizada na condicional simples para: if dados[i] < dados[i + 1]. Tente fazer isso em casa e veja se deu certo!

**1.3 COMPLEXIDADE DO *BUBBLE SORT***

Qual a complexidade Big-O da função do *Bubble Sort*? Como temos dois laços aninhados, mas as iterações do segundo laço não se alteram, temos uma PA constante e podemos fazer a propriedade da multiplicação. Sendo assim, o Big-O do ***Bubble Sort* é O(n²)**.

**1.4 MELHORANDO O *BUBBLE SORT***

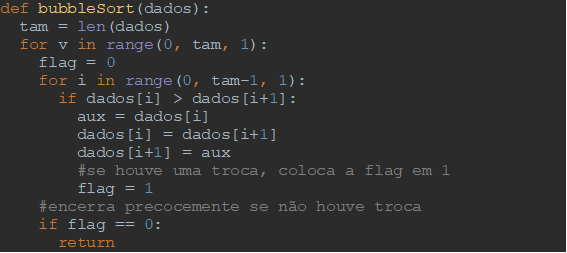
Relembre a Tabela 1. Você notou que nosso conjunto de dados já ficou ordenado logo no início da quarta varredura? Será que não podemos aprimorar esse algoritmo para identificarmos quando ele ficou ordenado e interromper o processo a qualquer momento, ganhando em desempenho? A resposta é sim, podemos.

O *Bubble Sort* que você viu até então apresenta complexidade para o pior caso, igual ao melhor, e é O(n²). Embora nossa principal análise ao longo desta caminhada seja para a complexidade do pior caso de nossos algoritmos, vamos parar por um instante para analisar o *Bubble Sort* no melhor caso também, para identificarmos como podemos melhorar esse algoritmo fazendo uma simples alteração.

Como já visto, o pior caso de um algoritmo é sempre aquele em que mais instruções são executadas. Sendo assim, o melhor caso é quando temos menos instruções executadas.

No *Bubble Sort* atual, se você passar como parâmetro para a função um conjunto de dados já ordenado (melhor caso), o algoritmo irá executar todas as iterações, gastando muito tempo desnecessário. Ou seja, o pior caso e o melhor caso são idênticos e igualmente ruins.

Podemos melhorar e corrigir isso inserindo uma variável que atue como uma *flag* no programa. Assim, essa *flag*pode identificar quando os dados já estão ordenados e interromper a ordenação precocemente, ganhando em desempenho. Veja a implementação dessa melhoria a seguir:



Assim, a complexidade do *Bubble Sort*melhorado é, para o pior caso, ainda O(n²), mas seu melhor caso agora fica somente Ω(n).

**TEMA 2 – MERGE SORT**

O **algoritmo de ordenação por intercalação**, ou ***Merge Sort****,*usufrui da estratégia de dividir para conquistar. O *Merge Sort*realiza a ordenação dividindo um conjunto de dados em metades iguais e reorganizando essas metades. É um algoritmo que opera de maneira recursiva, dividindo de maneira contínua o conjunto de dados até eles tornarem-se indivisíveis.

Sempre que duas metades estão ordenadas, um processo chamado de mesclagem (ou intercalação) é performado. A intercalação é um processo que pega dois conjuntos de dados ordenados e os combina em um só, ordenando, resultando em novo conjunto de dados. Vamos agora compreender o funcionando desse algoritmo mais detalhadamente.

**2.1 2.1 *MERGE SORT*: LÓGICA DE FUNCIONAMENTO**

Como temos um funcionamento pelo princípio de dividir para conquistar, precisamos entender qual é o problema que precisamos resolver e qual o caso-base.

Nosso problema consiste em ordenar um conjunto inteiro de dados, que em nossos exemplos serão listas em linguagem Python. Já nossos subproblemas, ou seja, aquele em que devemos dividir consiste em dividirmos o conjunto de dados sempre ao meio e ordenarmos as duas metades, atingindo o caso-base.

O primeiro passo do algoritmo de ordenação por intercalação é, portanto, encontrar o ponto central do nosso conjunto de dados, assim saberemos onde dividir nosso conjunto. Encontramos o meio do conjunto de dados pela equação:



Em que *início* é a posição inicial do conjunto de dados, *fim*é a posição final do conjunto de dados e *int* indica que devemos ficar somente com a parte inteira do resultado da divisão. Em linguagem Python, podemos escrever a equação anterior de uma maneira simplificada, como:



Em que *len* representa o tamanho do conjunto de dados sendo manipulado, e as duas barras // o resultado inteiro da divisão.

Com a equação anterior, **dividimos** os dados. Em seguida, **conquistamos**recursivamente os conjuntos de dados menores, até atingirmos os casos-base, que aqui corresponde à menor unidade indivisível de um conjunto de dados (conjunto menor do que dois elementos). Por fim, **combinamos** mesclando os subconjuntos de volta em um único conjunto.

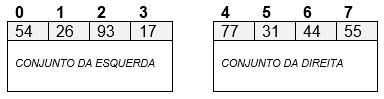
Queremos agora ordenar o conjunto desordenado de dados, contendo: . Vejamos o passo a passo:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| 54 | 26 | 93 | 17 | 77 | 31 | 44 | 55 |

Acima de cada número, temos o índice do conjunto de dados utilizando a nomenclatura da linguagem Python, que sempre inicia um conjunto de dados, como uma lista, no índice zero. Qual o ponto central para dividirmos esse conjunto?



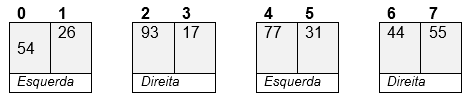
Quebraremos nosso conjunto de dados no índice 3, deixando 4 valores para cada lado:



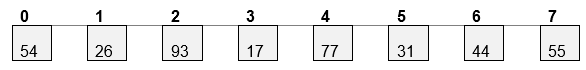
Só iremos parar de dividir quando nossos conjuntos tiverem tamanhos menores do que dois, certo? Portanto, continuamos e quebramos cada subconjunto em outros dois. Para tal, precisamos localizar o meio de cada um deles novamente:



Nossos quatro conjuntos, serão:

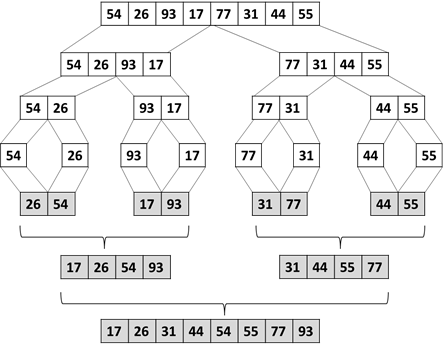


Temos que dividir novamente porque nossos conjuntos têm tamanho dois. Ficamos com:



Atingimos oito conjuntos indivisíveis e de tamanhos unitários. Agora, iremos mesclar os conjuntos de volta até atingir um único conjunto novamente. Faremos isso mesclando da mesma forma que dividimos. Ao mesclar, verificamos a ordem dos dados dos subconjuntos. A seguir, temos todo o processo de divisão e a posterior mesclagem dos dados do conjunto.

Figura 1 – *Merge Sort*: ordenando crescentemente o conjunto de dados:

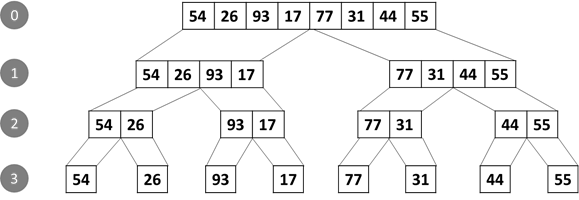


Note que quando atingimos as menores unidades de tamanho do nosso conjunto, iniciamos a grupá-las e ordená-las simultaneamente. Observe, por exemplo, do lado esquerdo os valores unitários 54 e 26. Ao serem mesclados, sua ordem já se altera, pois na ordenação crescente 26 é menor do que 54. Em seguida, mesclando o conjunto *[26, 54]* com *[17, 93]*, todos os quatro dados são verificados entre si e posicionados na ordem correta: *[17, 26, 54, 93]*.

Todas as divisões ocorrem de maneira recursiva e cada subconjunto é uma nova chamada. As funções recursivas, ao serem encerradas, vão originando os conjuntos ordenados de dados.

Observe que a quantidade de chamadas recursivas de um *Merge Sort* depende da quantidade de divisões que ocorreram. Conforme a Figura 2, houve-se a necessidade de quatro subdivisões até atingirmos o caso-base.

Figura 2 – *Merge Sort*: quantidade de níveis/subdivisões



Matematicamente, pode-se dizer que o total de conjuntos por nível é:

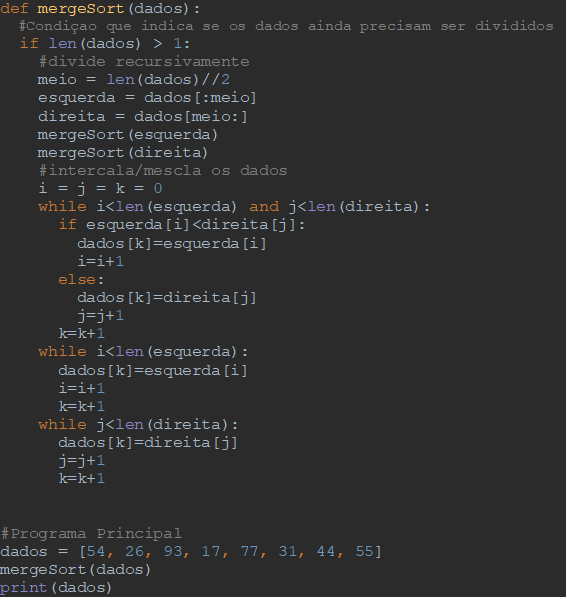
|  |  |
| --- | --- |
| **Nível** | **Total** |
| **0** |  |
| **1** |  |
| **2** |  |
| **3** |  |

Com isso, a quantidade total de chamadas recursivas da função *Merge Sort* será de:



**TEMA 3 – *MERGE SORT*: O ALGORITMO**

O *Merge Sort* precisa ser analisado com bastante calma, haja vista que seu algoritmo é um pouco extenso. A seguir, apresentamos este código.



O nosso algoritmo se inicia verificando se o tamanho do conjunto de dados é maior do que um. Caso seja maior do que um, significa que precisamos ficar dividindo os dados recursivamente, até essa condição não ser mais satisfeita.

Para dividirmos os dados, encontramos o ponto central (variável *meio*) e colocamos cada parte do conjunto em duas listas separadas (variáveis *esquerda*e *direita*). Note que, após dividido o conjunto, a função *mergeSort*é invocada novamente e ficamos nesse processo até obtermos os casos-base.

Em seguida, quando queremos mesclar os dados, fazemos um laço de repetição e dentro dele uma condicional simples. O objetivo desse laço é o de ir verificando a ordem dos elementos e colocando-os ordenados de volta dentro da variável da lista.

Após o primeiro laço, existem outros dois laços de repetição. Esses dois laços servem para preencher as lacunas que irão faltar de dados sobrando nos vetores.

**Saiba mais**

Como estão ordenados esse algoritmo de maneira decrescente?

A alteração ocorre somente em uma linha. Dentro do primeiro laço, temos uma condicional, certo? Experimente alterar o sinal de menor para o sinal de maior, ao comparar a parte esquerda com a direita e veja o que acontece.

**3.1 COMPLEXIDADE DO *MERGE SORT***

Qual a complexidade Big-O da função do *Merge Sort*? A análise aqui deve ser mais cautelosa do que o que fizemos no *Bubble Sort*. Lembre-se do princípio de funcionamento desse algoritmo. Ele opera com o princípio de dividir para conquistar, isso significa que teremos uma complexidade O(logn) atrelada a esse algoritmo. Porém, não paramos por aqui.

No algoritmo, após o dividir para conquistar, temos ainda laços de repetição posicionados um após o outro. Como os laços não estão aninhados, utilizamos o princípio da adição: O(n) + O(n) + O(n) = O(n). Por fim, utilizamos o princípio da multiplicação, para agregarmos as duas partes do algoritmo em uma só complexidade:



**TEMA 4 – QUICK SORT**

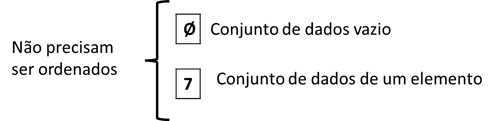
O algoritmo de ordenação rápida, ou *Quick Sort*, também usufrui da estratégia de dividir para conquistar como o seu irmão *Merge Sort*. O *Quick Sort* é talvez o algoritmo de ordenação mais famoso e também o mais utilizado entre todos, pois funciona muito bem para grande maioria dos casos de ordenação.

Todavia, a maneira como o *Quick Sort* trabalha com o dividir para conquistar é ligeiramente diferente, pois usufrui das características de desempenho dessa estratégia, mas sem precisar utilizar memória adicional como o *Merge Sort*. Como desvantagem perante a ordenação por intercalação, é que nem sempre o conjunto de dados será divido exatamente ao meio e, quando isso ocorrer, teremos um certo impacto no seu desempenho.

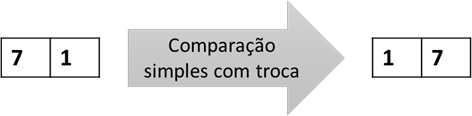
Vamos agora compreender o funcionando desse algoritmo mais detalhadamente.

**4.1 4.1 *QUICK SORT*: LÓGICA DE FUNCIONAMENTO**

Toda nossa análise será realizada para ordenação de dados crescente (do menor para o maior). Antes de sairmos ordenando um conjunto de dados com diversos elementos, vamos iniciar nossa análise por algo mais simples, certo? Qual é o conjunto de dados mais simples que um algoritmo de ordenação pode ordenar? Veja bem. Existem conjuntos que não precisam ser ordenados. Veja quais são:



Um conjunto de dados com dois elementos é bastante simples de ser ordenado. Basta verificar se o primeiro elemento é menor do que o segundo e trocá-los de lugar, caso necessário.



Até o momento, nenhuma dificuldade ou novidade, certo? Mas e um conjunto com três elementos? Como o *Quick Sort* ordenaria? Veja bem, esse algoritmo trabalha com dividir para conquistar. Portanto, precisamos atingir o caso-base. Eis a lógica desse algoritmo.

1. Escolhemos um elemento dentro do conjunto de dados, o qual chamamos de **pivô**. Em tese, o pivô pode ser qualquer elemento do conjunto. Porém, normalmente na literatura é escolhido o elemento central (Ascencio, 2030) ou então o primeiro elemento do conjunto (Bhargava, 2039).
2. Em seguida, realizamos um processo chamado de **particionamento**. Onde encontram-se os elementos menores do que o pivô e também os elementos maiores.
3. Com isso, particionamos o conjunto de dados em três partes. Em que deixamos do lado esquerdo os valores menores que o pivô, e a direita, os maiores.

*Quick Sort com 3 elementos*

Vejamos o exemplo a seguir com três elementos:



Vamos definir o pivô como sendo o primeiro elemento. Representaremos o pivô com um losango. Ao escolhermos o pivô, particionamos nosso conjunto de dados em dois. Como 10 é o primeiro elemento, ficamos com um conjunto vazio à esquerda:



Na sequência, o valor menor do que 10 é colocado à esquerda, e o valor maior do que 10, à direta. Ficamos com:



Nosso conjunto de dados foi ordenado usando *Quick Sort*:



*Quick Sort com 4 elementos*

Vejamos o exemplo a seguir com quatro elementos:



Vamos definir o pivô como sendo o primeiro elemento. Ficamos, portanto:

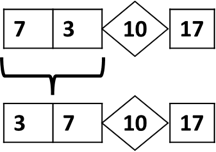


Os valores menores do que 10 são colocados à esquerda, e o valor maior do que 10, à direta. Ficamos com:



Observe que os valores à esquerda de 10 não estão ordenados. Isso porque o *Quick Sort* inicialmente só se preocupa em separar os valores para seus conjuntos, sem ordenar.

Agora, à esquerda do pivô ficamos com dois valores. Podemos então recursivamente ordenar esses valores, fazendo uma ordenação simples entre dois valores:



O resultado do vetor ordenado é:



*Quick Sort com cinco elementos*

Vejamos o exemplo a seguir com cinco elementos:



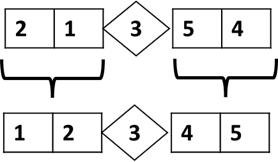
Vamos definir o pivô como sendo o primeiro elemento. Ficamos, portanto:



Dividindo os conjuntos:



Temos agora dois conjuntos de dois elementos para cada lado. Podemos fazer uma ordenação simples em cada lado:



O resultado do vetor ordenado é:



*Quick Sort com oito elementos*

Vamos ao desafio final. Vejamos o exemplo a seguir com nove elementos:



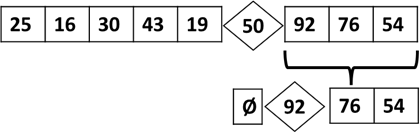
Vamos definir o pivô como sendo o primeiro elemento. Ficamos, portanto:



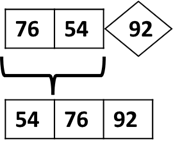
Dividimos os conjuntos:



Temos agora dois conjuntos de dois elementos para cada lado. Vamos precisar continuar dividindo os conjuntos até obtermos conjuntos de até dois dados, em que é possível ordenar. Vamos começar pelo lado direito:



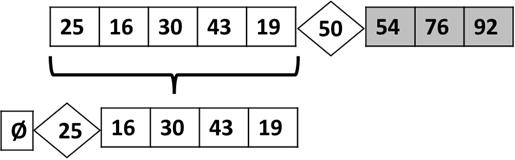
Definimos o pivô do conjunto de três dados como sendo novamente o primeiro. E, agora, conseguimos ordenar o que temos à direita, pois restaram somente dois dados:



Temos o conjunto do lado direito ordenado crescentemente. Deixaremos ele destacado em cinza na imagem e vamos trabalhar agora no lado esquerdo: definimos o pivô do conjunto de três dados como sendo novamente o primeiro. E, agora, conseguimos ordenar o que temos à direita, pois restaram somente dois dados:



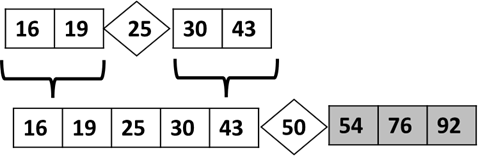
Note que, do lado esquerdo, temos agora um conjunto de cinco dados. Vamos então trabalhar ordenando esse conjunto como já vimos anteriormente. Definimos o pivô e dividimos o conjunto em duas partes:



Organizamos os valores menores do que 25, antes dele, e os maiores, após ele:



Nesse momento, ficamos com dois conjuntos de dois dados em ambos os lados. Se notarmos a imagem, os dados já estão ordenados. De todo modo, o algoritmo iria realizar essa verificação para confirmar a ordenação, portanto, teríamos:

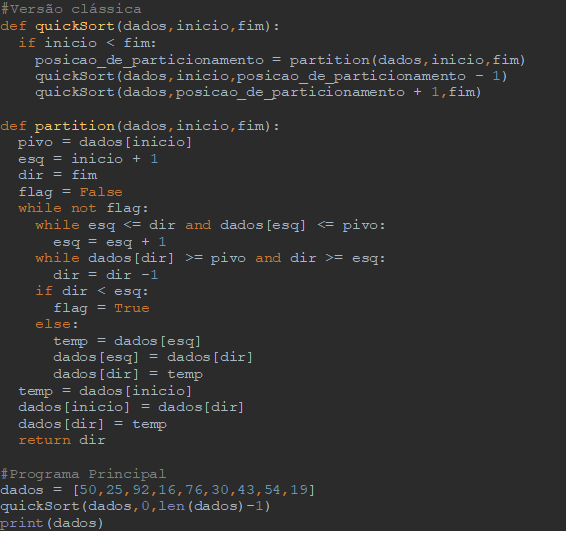


Desta forma, nosso conjunto agrupado novamente e ordenado crescentemente ficou:



**TEMA 5 – *QUICK SORT*: O ALGORITMO**

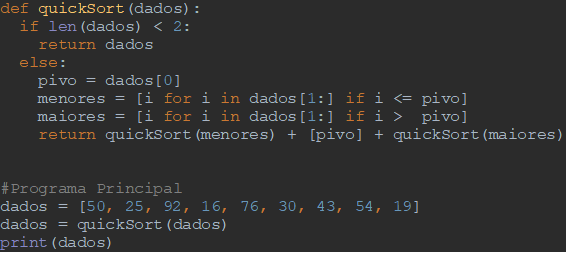
O *Quick Sort* precisa ser analisado com bastante calma, haja vista que seu algoritmo é um pouco extenso. A seguir, apresentamos esse código.



Note que, em nosso algoritmo, temos duas funções separadas. A primeira se chama *Quick Sort*. Ela é a responsável por ficar particionando os conjuntos de dados até atingirmos o caso-base, ou seja, cada conjunto de dados fica unitário.

A outra função chama-se *partition*(partição ou particionamento). Ela é responsável por realizar toda a separação e ordenação dos dados. Inicia-se com ela definindo o pivô como sendo o primeiro dado do conjunto. E, em seguida, varreduras são realizadas para que possamos separar os valores menores e maiores para seus respectivos lados.

Uma implementação alternativa do *Quick Sort* pode ser vista a seguir. A implementação a seguir parece ser bem mais simples do que a anterior, não concorda? Porém, ela só é possível de ser construída devido às características da linguagem Python, em que dentro de cada lista já podemos fazer as varreduras desejadas.



O nosso algoritmo se inicia verificando se o tamanho do conjunto de dados é menor do que dois. Caso seja, caímos em um caso que é indivisível (conjunto de dados vazio ou unitário) e, portanto, não precisamos ordenar.

Note que, para o pivô, é sempre atribuído o primeiro dado do conjunto (índice zero). Os valores menores e maiores são calculados na sequência. As variáveis maiores e menores estão sendo definidas em Python como uma lista contendo os respectivos valores.

No *return* existe uma recursividade. Ou seja, antes da função retornar algo ela invoca ela mesma para ambos os lados, esquerda e direita.

**Saiba mais**

Como, então, ordenamos esse algoritmo de maneira decrescente?

No segundo algoritmo, tente inverter os sinais de comparação das listas de maior e menor. E na abordagem clássica, como ficaria?

**5.1 COMPLEXIDADE DO *QUICK SORT***

Qual a complexidade Big-O da função do *Quick Sort*? A análise aqui deve ser mais cautelosa do que o que fizemos no *Bubble Sort.* Lembre-se do princípio de funcionamento desse algoritmo. Ele opera com o princípio de dividir para conquistar, isso significa que teremos uma complexidade O (logn) atrelada a esse algoritmo. Analisando o algoritmo clássico, em que o código está mais detalhado, percebemos que a função *quickSort* é quem de fato opera recursivamente.

Ainda temos a função *partition*. Ela contém dois laços de repetição aninhados, o que nos remete a uma PA e, portanto, O(n) \* O(n) = O(n²). Por fim, utilizamos o princípio da multiplicação para agregarmos as duas partes do algoritmo em uma só complexidade:



Observe que, na complexidade assintótica, ficamos somente com o termo de maior grau. Como *n²* cresce em grau muito mais do que *logn*, ficamos somente com O(n²).

**FINALIZANDO**

Nesta etapa, aprendemos três algoritmos de ordenação e suas respectivas complexidades. Veja o resumo comparativo.

* Bubble Sort: O(n²).
* Merge Sort: .
* Quick Sort: O(n²).

**REFERÊNCIAS**

ASCENCIO, A. F. G.; ARAÚJO, G. S. de. **Estruturas de Dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Halt 3, 2010.

BHARGAVA, A. Y. **Entendendo Algoritmos.**Novatec, 2017.

DROZDEK, A. **Estrutura de Dados e Algoritmos em C++**. Tradução da 4ª edição norte-americana. Cengage Learning Brasil, 2018.

KOFFMAN, E. B.; WOLFGANG, P. A. T. **Objetos, Abstração, Estrutura de Dados e Projeto Usando C++.** Grupo GEN, 2008.

**PROGRAMAÇÃO III**

AULA 3

Prof. Vinicius Pozzobon Borin

**CONVERSA INICIAL**

Está na hora de aprender uma nova estrutura de dados! Nesta etapa, você irá se deparar com as características de implementação de uma estrutura de dados denominada *lista encadeada*. As listas encadeadas são alocadas dinamicamente na memória e diferem das implementações de arrays encontrados em linguagens de programação, trazendo vantagens e desvantagens. Mas, atenção, uma lista em Python não é uma lista encadeada! Tenha isso em mente e você compreenderá melhor este motivo.

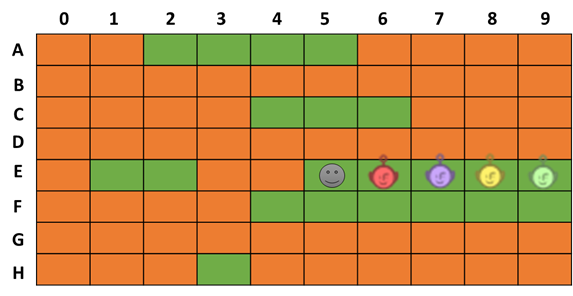
**TEMA 1 – ARRAYS E LISTAS ENCADEADAS**

Arrays, ou *vetores*, são estruturas de dados que trabalham com alocação de dados sequenciais na memória. Podemos classificar os arrays em estáticos e dinâmicos. Para melhor compreensão, veremos diferentes cenários.

**1.1 O PROBLEMA DOS ASSENTOS RESERVADOS NO CINEMA**

Você e mais quatro amigos (Tinky Winky, Dipsy, Laa-Laa, Poo) estão indo ao cinema, o qual só funciona com assentos reservados previamente. Vocês cinco gostariam de sentar juntos para aproveitar melhor o filme e, portanto, reservaram os assentos sequenciais do E5 até E9 (em verde). Então, todos se sentam juntos. Veja, a seguir, uma matriz que representa os assentos de cinema.

Figura 1 – Assentos no cinema. Em laranja, os ocupados; em verde, os livres.



A memória de um programa é dividida de maneira semelhante aos assentos de um cinema, pois cada bloquinho de memória irá representar um dado a ser armazenado. Quando precisamos armazenar um conjunto de dados, podemos organizá-lo a partir de um *array*, o que implica que todos os dados serão armazenados sequencialmente na memória do programa. Em nosso exemplo do cinema, você e seus amigos estão organizados como se fosse um array sequencial, pois estão todos juntos.

Porém, de última hora, mais um amigo seu, o Solzinho, decide aparecer. Agora, para sentarem juntos seria necessário que todos se movam para uma sequência de assentos que comporte todo mundo. A fila de trás (F4 até F9) seria a única possibilidade. Infelizmente, o cinema só trabalha com lugares reservados e vocês não podem simplesmente pular para a fileira de trás. Jonas, então, para não ficar sozinho, vai embora entristecido e sem ver o filme.

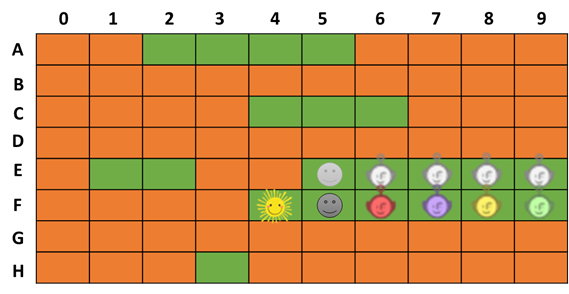
Essa situação em que temos mais um dado para ser colocado no array, mas sua inserção é negada, corresponde a um array do tipo estático, que nada mais é do que um conjunto de espaços sequenciais na memória, previamente reservado, mas que em hipótese alguma pode ter o seu tamanho alterado.

Voltando ao exemplo dos assentos, seria muito simples que você e seus amigos sentassem na fileira de trás, onde caberia todo mundo, mas isso foi negado a vocês, pois o array estático funciona com reserva prévia de espaços de memória. O inverso também é valido. Veja, você e seus amigos compraram cinco ingressos para ocupar assentos reservados. Caso alguém falte, o assento continuará pago e reservado, sem necessidade.

**1.2 O PROBLEMA DOS ASSENTOS SEM RESERVA NO CINEMA**

Vejamos agora outro cenário. Você e seus quatro amigos estão indo em outro cinema, mas neste não há exigência de reserva de lugares, ou seja, quem chegar por primeiro ocupa os lugares. Suponha que a configuração de espaços vazios é igual ao da Figura 1 e vocês cinco sentam-se nas cadeiras E5-E9. Novamente, de última hora, seu amigo Solzinho, atrasado, chega para assistir ao filme com vocês. Porém, agora não existe marcação de assentos. Neste caso, vocês simplesmente pulam para a fileira de trás, que comporta seis pessoas:

Figura 2 – Assentos no cinema sem reserva. Em laranja, os ocupados; em verde, os livres



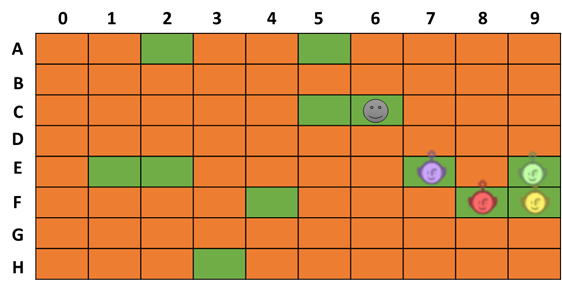
A analogia do cinema com assentos sem reserva, neste caso, corresponde a um array dinâmico, o qual também armazena dados sequencialmente. Agora, porém, à medida que surgem novos dados, é possível realocar o conjunto em outro espaço de memória em que caibam todos os dados.

Complementando o cenário, imagine que mais um amigo seu, além do Solzinho, chegou para assistir ao filme. Considerando que todos precisam sentar-se juntos, ele não poderia assistir ao filme, porque no exemplo da Figura 2 não existe nenhuma sequência de sete assentos livres.

**1.3 O PROBLEMA DOS AMIGOS DISTANTES NO CINEMA**

Vamos ao último cenário no cinema. Você e seus amigos quatro amigos agora vão ao cinema sem poltronas marcadas, mas os lugares próximos estão bastante escassos. Não existe nenhum lugar na sala de cinema com mais de dois lugares livres juntos. O filme é legal demais para ser perdido e vocês optam por assisti-lo mesmo separados fisicamente.

Figura 3 – Assentos no cinema separados. Em laranja, os ocupados; em verde, os livres



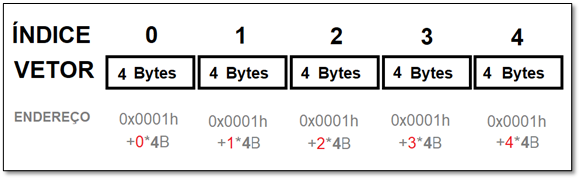
Solzinho, sempre atrasado, chega em cima do horário. Neste cenário, ele poderia facilmente sentar-se em qualquer outro local livre e assistir ao filme com seus amigos, na mesma sala de cinema. Esse tipo de configuração, de arranjo de dados sem necessidade de estarem sequencialmente alocados é chamado de *listas encadeadas*.

**1.4 ARRAYS E LISTAS ENCADEADAS: COMPARATIVO**

Arrays são conjuntos de dados alocados sequencialmente na memória do programa. Quando declaramos uma estrutura de vetor, na memória do programa ele é inicializado (alocado) a partir da primeira posição (endereço da primeira célula). Cada outra célula, a partir da segunda, possui um endereço de referência relativo à primeira célula endereçada. Este endereço é calculado considerando a posição da primeira célula, acrescido do tamanho em *bytes* de cada uma – tamanho este que depende do tipo de dado armazenado no vetor. Chamamos isto de *alocação sequencial*.

Observe, na Figura 4, um vetor de valores inteiros de dimensão, ou comprimento, igual a 5, ou seja, contendo 5 células. Sua inicialização na memória é dada pela primeira posição desse vetor, neste caso, no endereço *0x0001h*. Assumimos que o vetor homogêneo é do tipo inteiro de tamanho 4 *bytes* por célula. A segunda posição (célula) está alocada, portanto, na posição da memória 0x0001h + 4 *bytes*. A terceira posição (célula) está alocada, na posição da memória 0x0001h + 2\*4 *Bytes*. E assim por diante, até a última posição do vetor, o qual contém um tamanho fixo e conhecido previamente. A seguir, a Figura 4 representa graficamente a explicação.

Figura 4 – Endereçamento de uma alocação sequencial



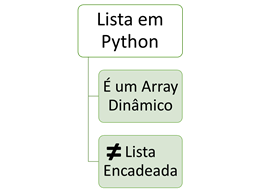
Podemos generalizar a forma de cálculo de cada posição na memória de um vetor pela equação 1:

|  |  |
| --- | --- |
|  | (1) |

em que  é o endereço conhecido da primeira posição do vetor, índice é a posição de cada célula e  é o tamanho de cada célula em *bytes,* neste exemplo, 4 *bytes.*

Diferentes linguagens de programação trabalham com maneira distintas de criar arrays estáticos e dinâmicos. A linguagem C/C++, por exemplo, só consegue criar arrays dinâmicos pela manipulação de ponteiros. Quando você declarar um vetor como *Tipo NomeDoVetor[quantidade\_de\_itens]*ele será estático. E atenção, não confunda! Na linguagem Python, a estrutura de lista (aquela que você cria assim: *Nome = []*) é, na verdade, um array dinâmico e não tem nenhum aspecto construtivo de uma lista encadeada.

Figura 5 – Lista em Python



O **array estático** recebe este nome quando um bloco na memória é destinado a ele previamente, isto é, mesmo que o bloco não seja usado inteiramente, ainda assim estará reservado e ocupando espaço no programa.

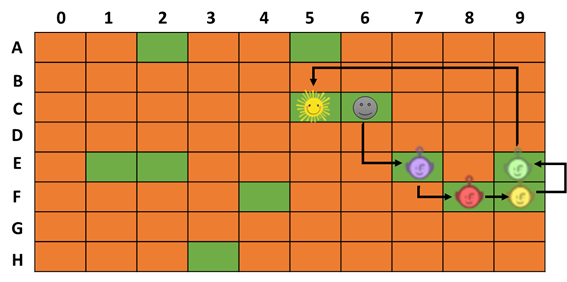
O **array dinâmico**é mais interessante, pois aloca somente o espaço sendo usado. Apesar disso, caso não exista na memória do programa nenhum bloco sequencial do tamanho desejado, haverá um problema, pois a alocação não acontecerá e o programa resultará em um erro de memória ou alocará o dado em local impróprio, podendo perder dados ou piorar o desempenho do programa.

A **lista encadeada**trabalhacom alocação não sequencial. Isso significa que os dados estão esparsos ao longo da memória do programa. Essa é uma característica muito interessante, pois, além de não usar memória em excesso, a lista encadeada é capaz utilizar quaisquer pequenos espaços livres na memória.

Mas se os dados estão esparsos na memória, como a estrutura de lista encadeada localiza seus elementos? Cada elemento da lista encadeada armazena na memória não só seus dados, mas também o endereço de onde está localizado o próximo elemento na memória.

Na Figura 6, temos a representação do cinema usando uma lista encadeada. Veja que foram colocadas flechas (ponteiros) de um elemento para outro. Comparando com o exemplo do cinema, é como se cada pessoa do grupo de amigos soubesse somente onde um de seus amigos está sentado, e o outro soubesse do próximo, e ninguém soubesse onde está todo mundo.

Figura 6 – Identificação do próximo elemento da lista encadeada



Então, se a lista encadeada é tão boa assim para alocação de memória, por que as linguagens ainda insistem em trabalhar com arrays? Veja bem: para toda vantagem, existe uma desvantagem. Em uma lista encadeada, cada elemento conhece somente o próximo elemento, pois armazena somente o endereço do próximo elemento – e isso pode gerar um certo problema.

Imagine que você recebeu um livro de cem páginas, e imagine também que cada página é o equivalente a um dado de um array. Você precisa chegar à página 50 para ler as informações nela contidas. Como cada página está numerada, basta abrir na página 50 e instantaneamente ler o que tem nela. O tempo para abrir na página 50, na 10 ou na 99, em tese, é o mesmo. Basta abrir o livro, pois as páginas estão numeradas. Um array funciona desta mesma maneira. Como alocamos um bloco sequencial de dados, o endereço de cada dado é conhecido pelo seu índice (veja a Figura 4 e a Equação 1).

Agora, imagine que nosso livro de 100 páginas armazena cada página em uma lista encadeada, na qual cada dado só sabe onde o próximo está. Ou seja, se queremos abrir na página 50, não sabemos onde ela está. Isso é o equivalente a tentarmos abrir um livro não numerado em uma página específica. E como chegamos em uma página que não tem número? Precisaremos contar uma a uma, a partir da primeira, até chegar à página desejada. Assim, uma lista encadeada apresenta um tempo de acesso em média inferior ao de um array.

**1.5 ARRAYS E LISTAS ENCADEADAS: COMPLEXIDADE**

Falamos sobre características, vantagens e desvantagens de arrays e listas encadeadas. Equal a complexidade de cada um? Em ambas as estruturas de dados podemos fazer distintas manipulações. Podemos ler um dado, fazer inserções em diferentes posições do conjunto de dados ou ainda deletar algum dado. A tabela a seguir apresenta o Big-O de algumas dessas manipulações.

Tabela 1 – Big-O de algumas manipulações

|  |  |  |
| --- | --- | --- |
| **Função** | **Array** | **Lista Encadeada** |
| **Leitura** | O(1) | O(n) |
| **Inserção no início** | O(n) | O(1) |
| **Inserção no fim** | O(n) | O(n) |
| **Inserção no meio** | O(n) | O(n) |

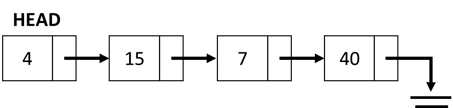
Um array, para ler um dado, independe do conjunto de dados, sendo O(1); a lista precisará varrer elemento por elemento até chegar no desejado, O(n).

**TEMA 2 – CONSTRUINDO LISTAS ENCADEADAS**

As listas encadeadas, também conhecidas como *listas ligadas*, ou ainda o termo em inglês *linked lists*, são compostas de um conjunto de dados em que cada dado é armazenado em um elemento que chamamos de *nó*, ou *nodo*. Cada elemento da lista é composto, portanto, minimamente de duas informações: o(s) dado(s) a serem armazenados e um endereço (ponteiro) na memória do próximo elemento da lista encadeada.

Chamamos o primeiro elemento da lista encadeada de *cabeça*, ou do seu termo em inglês, que é mais utilizado, *head*. A maneira de iniciar uma lista e localizar todos os seus valores é por meio do seu head. Podemos representar uma lista encadeada pelo desenho a seguir.

Figura 7 – Exemplo de lista encadeada com dados numéricos

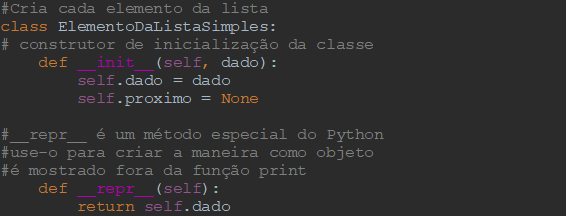


Note que o *head*contém o número 4 e aponta, ou seja, conhece o endereço do próximo elemento, o 15. Isso ocorre assim por diante, até chegarmos ao final desta lista, representada por um ponteiro nulo. Em suma as características das listas encadeadas são:

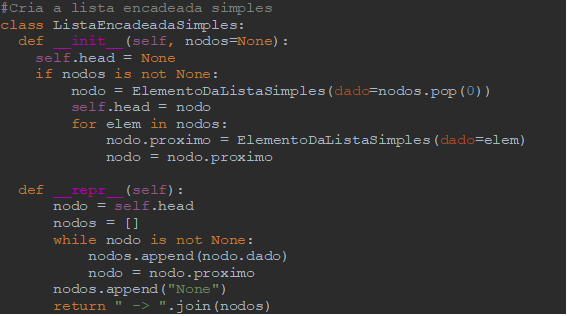
* sucessivos elementos conectados por ponteiros;
* o último elemento aponta para um endereço nulo, que é o final da lista;
* funciona dinamicamente, aumentando e diminuindo de tamanho conforme necessita;
* pode utilizar toda a memória destinada ao programa, uma vez que cada dado é capaz de ficar isolado na memória;
* não desperdiça memória, alocando somente o que precisa; e
* apresenta tempos de leitura Big-O de dados inferior aos arrays.

**2.1 LISTAS ENCADEADAS SIMPLES**

Existem algumas classificações das listas encadeadas, mas iremos focar nossos estudos, neste momento, na lista encadeada do tipo simples, em que cada elemento aponta para o próximo e o último, aponta para nulo, conforme a Figura 7. Implementaremos essa lista ligada em linguagem Python. Para isso, criaremos uma classe que representará cada elemento da lista encadeada. Veja:



Precisamos também criar uma segunda classe, que será, de fato, a lista encadeada. No constructo de inicialização (\_\_init\_\_) definimos que o *head*inicia-se vazio (*None*). Caso a lista não esteja vazia, temos:

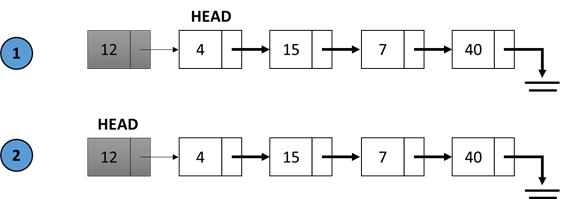


**2.2 INSERINDO NO INÍCIO DA LISTA ENCADEADA**

Suponha agora que temos uma lista encadeada já construída com alguns dados, mas queremos inserir mais um. A inserção pode ser feita no início da lista, no meio, ou no final; inserir no início da lista significa inserir antes do *head*atual. Para tal, precisamos respeitar duas etapas na inserção.

Antes da inserção, é valido observar que primeiro criamos o novo elemento na memória, instanciando-o pela classe *ElementoDaListaSimples*. Em seguida, a primeira etapa de inserção corresponde a fazer este novo elemento apontar para o atual *head*. Na segunda etapa, transformamos o novo elemento, no novo *head*, conforme a Figura 8.

Figura 8 – Etapas de inserção no início da lista encadeada



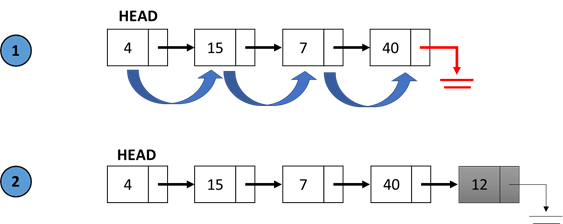
O código de inserção no início da lista é apresentado a seguir.



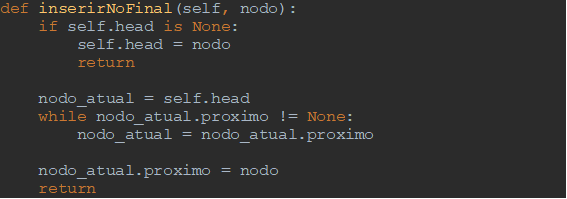
**2.3 INSERINDO NO FINAL DA LISTA ENCADEADA**

Podemos também inserir um dado no final da lista encadeada, isto é, após o elemento que contém o ponteiro nulo. Antes da inserção, é valido observar que primeiro criamos o novo elemento na memória, instanciando-o pela classe *ElementoDaListaSimples.* Em seguida, a primeira etapa de inserção corresponde a uma varredura na lista até localizarmos um ponteiro nulo na lista. Lembre-se de que na lista só conhecemos o *head* e que o *head*conhece o próximo elemento, e assim por diante, até chegarmos ao final*.*Uma vez localizado o ponteiro nulo, fazemos ele apontar para o novo elemento de nossa lista encadeada, conforme a Figura 9.

Figura 9 – Etapas de inserção no final da lista encadeada



O código a seguir realiza essa inserção ao final da lista ligada. Note o laço de repetição que só se encerra quando localizamos um endereço nulo (*None* no Python). É válido observar também que logo no início do código temos um teste condicional para verificar se nossa *linked list* é, ou não, completamente vazia. Caso ela esteja vazia, significa que o *head*está vazio e, portanto, ao invés de varrermos podemos simplesmente inserir o dado no *head.*



**2.4 CLASSIFICAÇÃO DE LISTAS ENCADEADAS**

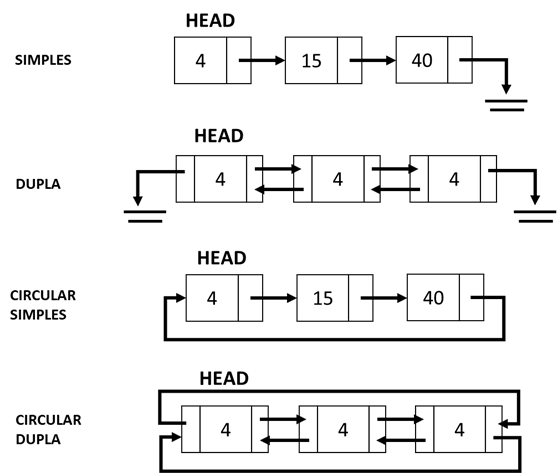
Existem diferentes tipos de listas encadeadas. Vamos conhecer um pouco mais sobre cada uma delas na teoria, embora estejamos focando nossos estudos somente na lista simples.

Tabela 2 – Tipos de lista encadeada

|  |  |  |
| --- | --- | --- |
| **Tipo** | **Endereços armazenados** | **Último elemento** |
| **Simples** | Próximo | Aponta para nulo |
| **Dupla** | Próximo e anterior | Aponta para nulo |
| **Simples e circular** | Próximo | Aponta de volta para o *head* |
| **Dupla e circular** | Próximo e anterior | Aponta de volta para o *head* |

Note que a diferença entre uma lista simples e uma dupla é a quantidade de endereço conhecidos; na dupla é possível ir e voltar na lista. Já o que difere ambas de listas circulares é que em uma circula o último elemento da lista, que aponta de volta para o início da mesma, não ficando nulo.

Figura 10 – Tipos de listas encadeadas representadas graficamente



**TEMA 3 – PILHAS (*STACKS*)**

Uma estrutura de dados do tipo pilha, ou *stack*, não é exatamente uma nova estrutura de dados, mas, sim, uma maneira de operar dados dentro de uma estrutura em programação. Vamos compreender o conceito de pilha com um exemplo de pilhas de objetos.

Imagine que você está indo à academia e está organizando anilhas de pesos de 10 Kg e 20 Kg; você precisa empilhar todos esses pesos. Para isso, você pega uma anilha de 20 Kg e coloca no chão. Em seguida, pega outra anilha de 20 Kg e coloca em cima da anterior, e assim por diante. Então, você vai pegando anilha por anilha e empilhando, formando uma pilha de anilhadas bastante pesadas. Veja a imagem a seguir.

Figura 10 – Pilha com anilhas

Crédito: Ilja Generalov / Shutterstock.

Mais tarde, outra pessoa veio à academia e decidiu utilizar suas anilhas que estavam empilhadas. Porém, a pessoa estava precisando das anilhas de 20 Kg, justamente as que estavam posicionadas mais abaixo da pilha. Para obter seu acesso, a pessoa precisou então desempilhar anilha por anilha até chegar à desejada. Aliás, imagine tentar puxar as anilhas mais abaixo sem remover as de cima? Haja força para isso!

Em resumo, uma estrutura de dados funcionando como pilha, funciona como uma pilha de anilhas. Uma estrutura de dados é uma pilha quando só conseguimos manipular o que está em seu topo. Ou seja, quando empilhamos as anilhas, estamos empilhando em cima do topo, e quando removemos as anilhas, desempilhamos somente o que está no topo. Nunca, em hipótese alguma, podemos inserir ou remover anilhas que estão no meio.

Uma estrutura de pilha em programação opera com o princípio chamado *o primeiro que entra é o último que sai*. A expressão correspondente em inglês é *first in last out (filo)***[[1]](https://conteudosdigitais.uninter.com/libraries/newrota/?c=/gradNova/2023/bachareladoEngSoftware/programacaoIii/a3&hash=3u0Jf1zuk93zpxVYIcnLAlZYm7U9Lk9Zlv+zz27MYdX/qhtrkmlbiPztyA5NjeB24iiNG/PbRJIOC+UdHAgVUKG31RKdwEZUuAYdr6rYQ0H5Ejenwpo9uneFd6awkPUXEZDHJp+5r8Dr2q4ROS9rfiEfQAPk26gK8KJl4FHVCGE=&ne=False" \l "_ftn1" \o ")***.*As estruturas de pilhas apresentam inúmeras aplicações em programação. Primeiro, qualquer algoritmo recursivo trabalha com uma estrutura de pilha, porque a recursividade opera empilhando funções, e somente a função mais ao topo da pilha é que está em execução. Sistemas operacionais trabalham muito com estruturas de pilhas internamente para funcionarem corretamente também.

Mas quando uso uma pilha em meu código? Vejamos alguns exemplos. Primeiro, podemos citar um exemplo clássico de implementação com pilhas que é o cálculo de expressões matemáticas. Imagine que seu algoritmo precisa validar e resolver uma equação contendo somas, subtrações, multiplicações e divisões – tudo isso ainda dentro de parênteses. Podemos criar um algoritmo que analisa a equação e a ordem dos operadores e vai empilhando todas as operações na ordem em que elas precisam ser executadas, bastando na sequência desempilhar e operar.

Outros exemplos seriam algoritmos de cálculos de conversão de base (decimal para binário, por exemplo), ou ainda podemos construir um jogo que contenha recursos de pilhas, como o jogo de cartas *FreeCell*, em que precisamos criar diferentes pilhas de cartas para manipular. Ferrari (2014) ensina a implementar o *FreeCell*com pilhas (e diversos outros jogos).

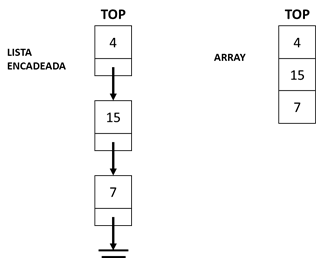
**3.1 CONSTRUINDO E MANIPULANDO UMA PILHA**

A construção de pilhas pode ser realizada utilizando algumas estruturas de dados já conhecidas. Por exemplo, podemos construir pilhas empregando arrays dinâmicos (lista em Python) ou com listas encadeadas. Isso é possível justamente porque uma pilha nada mais é do que a maneira como inserimos e removemos dados na estrutura, já uma array ou uma lista encadeada define como os dados são organizados na memória; ambos se complementam.

Na Figura 11, temos graficamente duas representações de pilhas: uma envolvendo listas encadeadas e outra, os arrays. Note que em ambas o primeiro elemento é sempre chamado de *topo*, ou*top*, que é a única posição que conseguimos manipular. Mas, em um array ou uma lista encadeada, é possível manipular qualquer posição? Como isso será uma pilha? Veja bem, certamente é possível manipular qualquer posição. Todavia, para que essa manipulação seja caracterizada como uma pilha, só podemos, obrigatoriamente, manipular seu topo. Qualquer outro tipo de manipulação descaracteriza a estrutura de pilha.

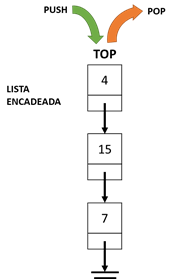
Se implementar a pilha como uma lista encadeada, o *top* seria o equivalente ao *head*, em que só poderíamos inserir ou remover dados diretamente do *head*. Ou seja, implementar uma pilha com listas encadeadas significa fazer uma inserção ou remoção somente no início dela.

Figura 11 – Pilhas representadas com listas encadeadas e com arrays



Acostume-se com os termos! A inserção em uma pilha é chamada de *empilhar*, ou em inglês, *push*. A remoção em uma pilha é *desempilhar* ou *pop*.

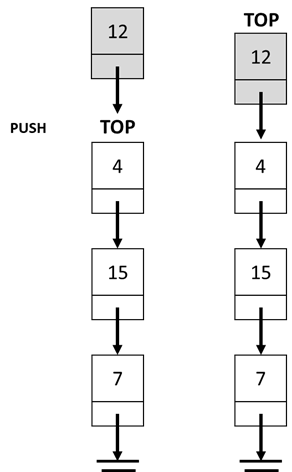
Figura 12 – *Push* e *pop* em pilhas com listas encadeadas



Ao fazer o *push*, se estivermos manipulando uma lista encadeada, primeiro iremos alocar o novo elemento dentro da memória do programa. Em seguida, este novo elemento irá apontar para o *top* da lista (*head*), e então transformamos o novo elemento no novo *top* (*head*) da lista ligada.

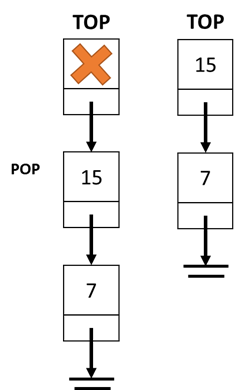
Percebeu que a sequência de passos para inserir na pilha é a mesmo para inserir no início de uma lista encadeada? De fato é o mesmo, só alteramos a terminologia de *head* para *top*. Na Figura 13 temos as duas etapas de inserção.

Figura 13 – Passo a passo do *push* com listas encadeadas



Ao fazer o *pop*, se estivermos manipulando uma lista encadeada, iremos sempre remover o elemento colocado no *top*. Para isso, fazemos o próximo elemento virar o *top*, e em seguida apagamos o elemento antigo do *top*.

Figura 14 – Passo a passo do *pop* com listas encadeadas



**TEMA 4 – FILAS (*QUEUES*)**

Uma estrutura de dados do tipo fila, ou *queue*, é uma maneira bastante particular de operar dados na programação. Vamos compreender o conceito de fila com um exemplo de fila de pessoas: imagine que o supermercado do seu bairro está fazendo uma mega promoção de queima de estoque. É claro que você não quer perder nenhum desconto, certo? Portanto, você decide chegar ao supermercado antes que ele abra, para ser o primeiro da fila. Você chega uma hora antes da abertura e fica na primeira posição da fila. Com o passar do tempo, outras pessoas começam a chegar. Naturalmente, todos que chegam depois de vocês entram ao final da fila.

Figura 15 – Fila de supermercado

Crédito: MikeDotta / Shutterstock.

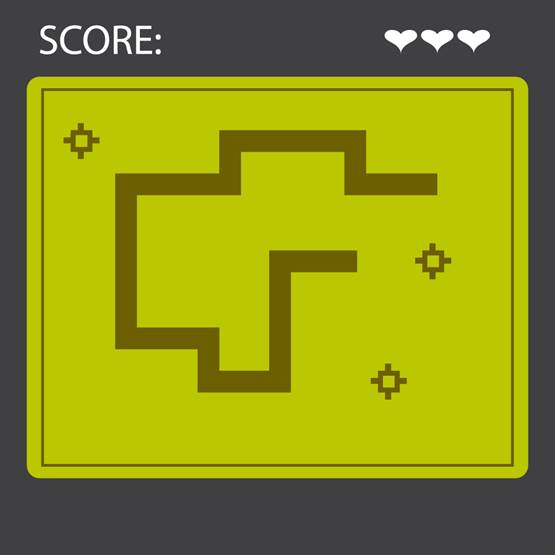
Ao abrir as portas do mercado, quem será o primeiro a entrar? Você! Note que, em uma fila, sempre quem chega primeiro é atendido primeiro, e quem chega por último, entra ao final da fila e será atendido por último. Uma estrutura de dados operando como uma fila, opera com o princípio de *o primeiro que entra é o primeiro que sai*, ou em inglês, *first in first out* (*fifo*)[[2]](https://conteudosdigitais.uninter.com/libraries/newrota/?c=/gradNova/2023/bachareladoEngSoftware/programacaoIii/a3&hash=3u0Jf1zuk93zpxVYIcnLAlZYm7U9Lk9Zlv+zz27MYdX/qhtrkmlbiPztyA5NjeB24iiNG/PbRJIOC+UdHAgVUKG31RKdwEZUuAYdr6rYQ0H5Ejenwpo9uneFd6awkPUXEZDHJp+5r8Dr2q4ROS9rfiEfQAPk26gK8KJl4FHVCGE=&ne=False" \l "_ftn2" \o ").

Em resumo, um conjunto de dados (pessoas!) funciona como uma fila que sempre só inserirmos no final do conjunto de dados (pessoa entram no final da fila) e sempre que só removermos do início do conjunto de dados (chegou primeiro na fila, é atendido primeiro). Nunca, em hipótese alguma, podemos inserir ou remover dados de maneiras distintas das citadas.

Como aplicações de algoritmos de filas, podemos citar a fila de impressão. Sabe quando você está enviando algum documento para ser impresso em uma impressora e outra pessoa também está enviando, simultaneamente? O algoritmo da fila de impressão irá atender à demanda e imprimir primeiro o arquivo que chegar antes. Chegou primeiro, imprimiu primeiro, e os outros vão sendo colocados ao final da fila. A área de redes de computadores manipula o tráfego da rede por meio de teoria de filas. Um *player* de música pode ter sua fila de reprodução de músicas implementada dessa maneira.

E sabe o famoso jogo da cobrinha (*snake*), que fez muito sucesso em celulares 15 anos atrás? Pois bem, ele pode ser implementado com uma estrutura de fila. A cobra será a fila e cada vez que comer um novo bloquinho, este bloco é anexado ao final da cobra, aumentando sua cauda.

Figura 16 – Jogo *snake*

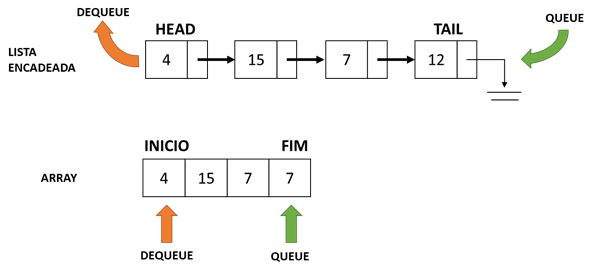
Crédito: 2DAssets / Shutterstock.

**4.1 CONSTRUINDO E MANIPULANDO UMA FILA**

A construção de filas pode ser realizada utilizando algumas das estruturas de dados já conhecidas. Por exemplo, podemos construir pilhas empregando arrays dinâmicos (lista em Python) ou com listas encadeadas.

Na Figura 17, temos graficamente duas representações de filas. Uma envolvendo listas encadeadas, e outra os arrays. Se implementarmos a fila como uma lista encadeada, iremos manipular seu *head*, e só poderíamos remover dados diretamente do *head*. Já a inserção na fila ocorre sempre ao seu final, por isso temos o costume de já armazenar em outra variável esse último elemento, para facilitar seu acesso. Na lista, chamamos o último elemento de *cauda*, ou *tail*. Ou seja, implementar uma fila significa fazer uma inserção (*queue*) no final dela, e fazer a remoção (*dequeue*) no início dela.

Figura 17 – *Queue* e *dequeue* em filas com listas encadeadas e com arrays

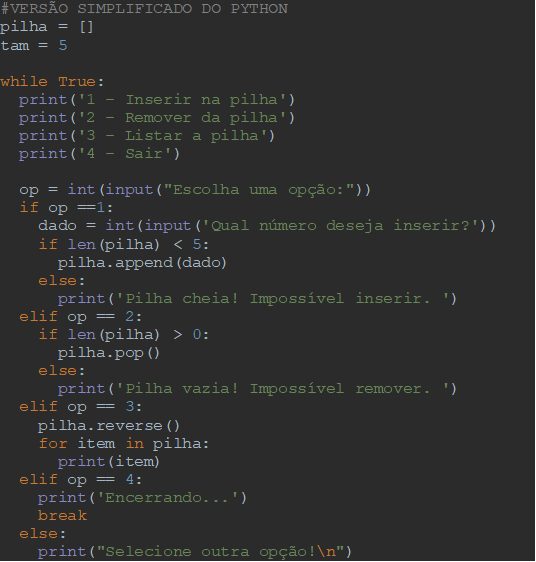


**4.2 IMPLEMENTANDO PILHAS E FILAS**

Precisamos agora verificar o código e aprender a implementação de ambos os códigos, de pilhas e de filas, em linguagem Python. As implementações apresentadas neste momento correspondem às arrays dinâmicas (listas em Python). Iremos verificar a implementação simplificada que usufrui das características da linguagem Python, facilitando nosso aprendizado.

**4.3 CÓDIGO DA PILHA**

O código de uma pilha implementada em Python pode ser verificado a seguir. O programa foi construído com listas em Python.



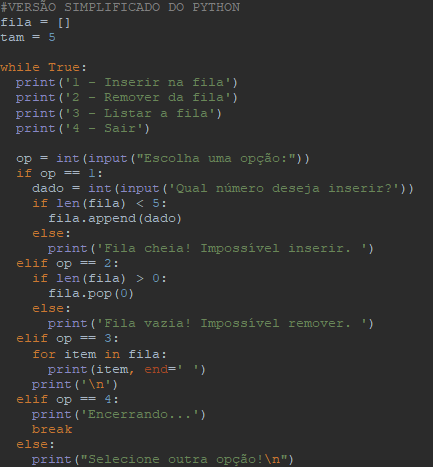
Note que criamos um código que contém um menu. Nele, o usuário é capaz de inserir um dado numérico na pilha, remover, mostrar a pilha na tela ou encerrar o programa. Como estamos trabalhando com uma lista em Python, podemos utilizar um método existente na linguagem chamado de *append*, que sempre adiciona (faz um apêndice) de um dado após todos os outros. Assim, entendemos que o topo de nossa pilha é sempre o último elemento da lista.

Para fazer a remoção de um dado na pilha, existe outro método, chamado de *pop*. Ele remove sempre o dado de um determinado índice passado como parâmetro. De acordo com a sintaxe da função, caso não passemos nenhum índice como parâmetro, o padrão é o índice -1, ou seja, sempre o último elemento, que é exatamente nosso topo da pilha. Portanto, inserimos no início com *append* e removemos do início com *pop(),*ou *pop (-1)*.

**4.4 CÓDIGO DA FILA**

O código de uma fila implementada em Python pode ser verificado a seguir. Foi criado um código que contém um menu, no qual o usuário pode inserir um dado numérico na fila, remover, mostrar a fila na tela ou encerrar o programa. Como estamos trabalhando com uma lista em Python, podemos utilizar um método existente na linguagem, o *append*, que sempre adiciona (faz apêndice) de um dado após todos os outros. Diferente da pilha, podemos entender que o final de nossa fila é sempre o último elemento da lista.

Para fazer a remoção de um dado na pilha, existe outro método, o *pop*, que remove sempre o dado de um determinado índice passado como parâmetro. Na pilha, havíamos usado o valor padrão de índice, que era o -1. Agora iremos propositalmente passar como parâmetro o valor zero no *pop*. Assim, removeremos sempre o primeiro dado da lista. Portanto, inserimos no final com *append* e removemos do início com *pop(0)*.



**FINALIZANDO**

Ao longo desta etapa estudamos novas estruturas de dados. Aprendemos as listas encadeadas, que são estruturas de dados com alocação dinâmica e não sequencial, onde cada elemento da lista contém seus respectivos dados e também o endereço para o próximo elemento desta lista.

A lista encadeada contém como principal vantagem a capacidade de ser alocada dinamicamente na memória, e como desvantagem o seu desempenho no ato da leitura dos dados, pois depende do tamanho do conjunto de dados, diferentemente de um array alocado sequencialmente.

Investigamos também as pilhas (*stacks*) e as filas (*queues*). Uma pilha opera com o princípio de FILO, enquanto as filas operam com FIFO. Ambas apresentam a remoção dos dados sempre em seu início (ou topo), enquanto a inserção na pilha ocorre no início (topo) e na fila ao final.

**REFERÊNCIAS**

ASCENCIO, A. F. G.; ARAÚJO, G. S. **Estruturas de Dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Halt 3, 2010.

BHARGAVA, A. Y.; **Entendendo Algoritmos**. Novatec, 2017.

DROZDEK, A. **Estrutura de Dados e Algoritmos em C++**. 4 ed.edição norte-americana (tradução). Cengage Learning Brasil, 2018.

FERRARI, R. *et al*. **Estruturas de Dados com Jogos**. Elsevier, 2014.

KOFFMAN, E. B.; WOLFGANG, P. A. T. **Objetos, Abstração, Estrutura de Dados e Projeto Usando C++**. Grupo GEN, 2008.

[[1]](https://conteudosdigitais.uninter.com/libraries/newrota/?c=/gradNova/2023/bachareladoEngSoftware/programacaoIii/a3&hash=3u0Jf1zuk93zpxVYIcnLAlZYm7U9Lk9Zlv+zz27MYdX/qhtrkmlbiPztyA5NjeB24iiNG/PbRJIOC+UdHAgVUKG31RKdwEZUuAYdr6rYQ0H5Ejenwpo9uneFd6awkPUXEZDHJp+5r8Dr2q4ROS9rfiEfQAPk26gK8KJl4FHVCGE=&ne=False" \l "_ftnref1" \o ") Dependendo da literatura, é possível encontrar esta expressão ao contrário: o último que sai é o primeiro que entra, ou *last out first in (lofi)*.

[[2]](https://conteudosdigitais.uninter.com/libraries/newrota/?c=/gradNova/2023/bachareladoEngSoftware/programacaoIii/a3&hash=3u0Jf1zuk93zpxVYIcnLAlZYm7U9Lk9Zlv+zz27MYdX/qhtrkmlbiPztyA5NjeB24iiNG/PbRJIOC+UdHAgVUKG31RKdwEZUuAYdr6rYQ0H5Ejenwpo9uneFd6awkPUXEZDHJp+5r8Dr2q4ROS9rfiEfQAPk26gK8KJl4FHVCGE=&ne=False" \l "_ftnref2" \o ") Dependendo da literatura, é possível encontrar esta expressão ao contrário: *o primeiro que sai é o primeiro que entra*, ou *first out first in (fofi)*.

**PROGRAMAÇÃO III**

AULA 4

Prof. Vinicius Pozzobon Borin

**CONVERSA INICIAL**

Aqui, você irá investigar e aprender a manipular estruturas não lineares de dados estudando a estrutura de árvore binária. Essa estrutura tem como principal aplicação operar em sistemas de busca, pois é construída de tal forma que facilita a localização dos seus elementos.

O objetivo desta etapa é apresentar os conceitos que envolvem a estrutura de dados do tipo árvore. Ao longo deste documento, será conceituada a estrutura de uma árvore e serão investigados dois tipos bastante comuns de árvores. Será apresentado como realizar as manipulações dos dados dentro de uma estrutura de árvore, como a inserção e a remoção dos dados. Os tipos de árvores estudadas serão:

* 'Árvore binária; e
* 'Árvore de Adelson-Velsky e Landis (árvore binária balanceada).

**TEMA 1 – ÁRVORES BINÁRIAS**

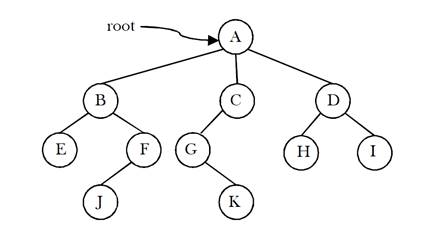
Imagine que você é responsável por implementar um vasto sistema de cadastro e armazenamento de dados de usuário de uma rede social. A quantidade de usuários será imensa, certo? Você precisa implementar uma estrutura de dados que seja eficiente na busca dentro desta rede, caso contrário, localizar um dado tomará muito tempo.

Uma opção inicial e que talvez esteja pensando é implementar um *array*e trabalhar com o algoritmo de busca binária. Caso siga por esse caminho, verá que precisará de um *array* longo demais. Além disso, caso precise inserir um novo dado nesse *array*, terá um tremendo processamento matemático para colocar os dados na posição certa, gastando processamento e memória.

E se você implementar a solução com uma lista encadeada, resolveria o problema? Também não. Lista encadeadas resolveriam o problema de inserção de um dado em qualquer posição, mas não são nada eficientes quando se trata de busca de dados. Isso porque uma lista encadeada é linear, e cada elemento só conhece seu sucessor (e anteceder se for uma lista dupla).

**Uma possível solução para esse tipo de problema é o emprego de uma estrutura não linear, a árvore, como uma árvore binária de busca**. Uma estrutura de dados é denominada de árvore quando seus elementos criam ramificações na estrutura, gerando **subárvores**. Esses ramos podemos chamar também de galhos. A Figura 1 ilustra um exemplo de uma árvore ternária, ou seja, em que um elemento pode se ramificar em até três outros. Existem diferentes classificações para árvores e, de um modo geral, podemos chamá-las de **árvores N-árias**.

Figura 1 – Exemplo de uma estrutura de árvore, neste caso, ternária



**1.1 ÁRVORES BINÁRIAS E SUAS PROPRIEDADES**

Uma lista encadeada apresenta uma característica importante: cada elemento da lista tem acesso somente ao próximo elemento (lista simples) ou ao próximo e ao anterior (lista dupla). Todos os elementos dessa lista são organizados de tal forma que é necessário percorrer, fazer uma varredura, da estrutura de dados para se ter acesso ao elemento desejado. Chamamos uma **lista encadeada** de uma estrutura de dados com **organização linear**.

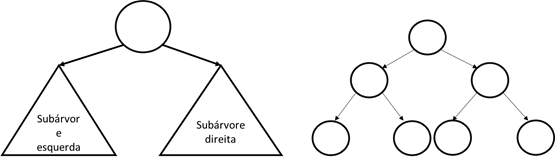
**As árvores, como uma árvore binária, são estrutura de dados não lineares**, em que são organizadas com elementos que não estão, necessariamente, encadeados, formando ramificações e subdivisões na organização da estrutura de dados.

A árvore binária é um tipo de árvore e é também a mais simples de ser estudada. Nesta etapa, focaremos somente nossos estudos em árvores binarias, as quais apresentam algumas características distintas, mesmo se comparando com outros tipos de árvores. Vamos analisar algumas dessas características e aproveitar para conceituar alguns termos próprios referentes a árvores:

* Nó raiz (*root*) – Nó original da árvore. Todos derivam dele;
* Nó pai – Nó que dá origem (está acima) de pelo um outro nó;
* Nó filho – Nó que deriva de um nó pai; e
* Nó folha/terminal – Nó que não contém filhos.

**Uma árvore é chamada de binária quando cada elemento (nó/nodo) da árvore contém, no máximo, dois (bi) filhos.**Cada elemento da árvore pode gerar até duas subárvores. A partir da raiz, podemos dizer que ela irá criar duas ramificações. Chamaremos uma de subárvore esquerda, e a outra de subárvore direita, conforme Figura 2.

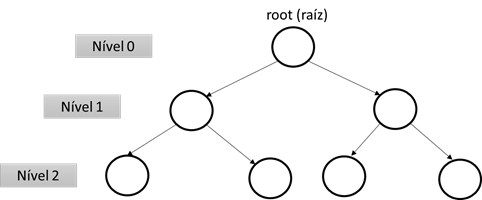
Figura 2 – Árvore binária e suas subárvores



**1.2 NÍVEIS, ALTURA E PROFUNDIDADE DE ÁRVORES BINÁRIAS**

Cada nova ramificação de uma árvore representa um nível. Dizemos que a raiz é sempre o nível 0. A partir dela, cada nova geração de filhos recebe um nível a mais, veja na Figura 3.

Figura 3 – Níveis na árvore binária



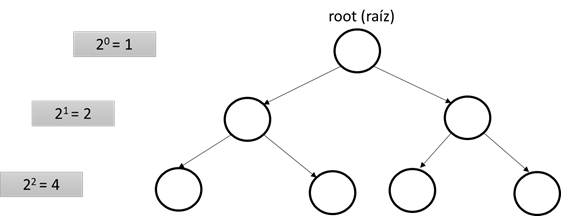
O conceito de nível de árvore é importante, pois eventualmente precisamos saber a profundidade de uma árvore binária (para fins de balanceamento, por exemplo, como veremos ainda nesta etapa). Para encontrarmos a **profundidade** de um nó da árvore, basta sabermos o nível daquele nó na árvore em relação à nossa raiz. Exemplo, se desejarmos saber a profundidade de um elemento no nível 2 da árvore, fazemos: 2 – 0 = 2. Portanto, a profundidade é 2.

O conceito de altura é similar à profundidade. A diferença é que a **altura**pode ser calculada pela diferença de nível entre dois nós. Por exemplo, a altura de um elemento no nível 2 da árvore, em relação à raiz, é 2 – 0 = 2, mesmo valor da profundidade. Agora, a diferença entre um elemento no nível 2, em relação ao nível 1, é 2 – 1 = 1.

**1.3 TOTAL DE ELEMENTOS POR NÍVEL NA ÁRVORE BINÁRIA**

Qual a quantidade máxima de elemento em uma árvore binaria, considerando que sabemos quantos níveis ela tem? Vejamos a Figura 4. Por se tratar de uma árvore binária, a cada nova ramificação, dobramos a quantidade máxima de elementos do nível anterior.

Figura 4 – Total de elementos por nível na árvore binária



Se nossa árvore tiver somente uma raiz, temos um só elemento. Se a árvore tiver um segundo nível, teremos mais dois elementos, portanto, três no total. Se a árvore tiver um terceiro nível, teremos mais quatro elementos, portanto, sete no total. Vejamos a tabela a seguir que resume para nós essa explicação.

|  |  |  |
| --- | --- | --- |
| **Nível** | **Nós no nível** | **Total de elemento por profundidade** |
| **0** | 20 = 1 | 21 - 1 = 1 |
| **1** | 21 = 2 | 22 - 1 = 3 |
| **2** | 22 = 4 | 23 - 1 = 7 |
| **3** | 23 = 8 | 24 - 1 = 15 |
| **4** | 24 = 16 | 25 - 1 = 31 |
| **5** | 25 = 32 | 26 - 1 = 63 |

Note que, para sabermos o total de elementos possível em uma árvore binária completa, podemos generalizar o cálculo da terceira coluna da tabela para:

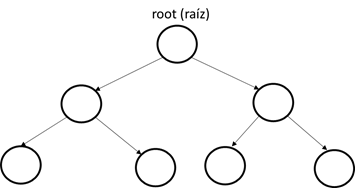


**1.4 TIPOS DE ÁRVORES BINÁRIAS**

Precisamos definir algumas nomenclaturas para árvores binárias. Uma **árvore estritamente binária** é aquela em que cada nó contém, sempre, exatamente dois filhos, ou nenhum. Uma **árvore binária completa** é aquela em que todos os nós contendo menos de dois filhos estão colocados somente no último ou no penúltimo nível da árvore.

A **árvore binária cheia** é assim denominada quando cada nó tem exatamente dois filhos, e os nós folhas estão sempre no mesmo nível. Ou seja, é quando a árvore é estritamente binária e completa ao mesmo tempo. A Figura 5 apresenta uma árvore binária cheia.

Figura 5 – Árvore binária cheia



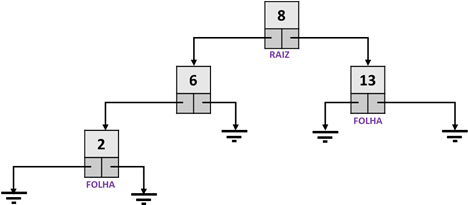
**TEMA 2 – ÁRVORE BINÁRIA DE BUSCA**

Para que uma árvore binária opere de maneira mais eficiente para busca, precisamos cadastrar seus dados de maneira organizada para facilitar a busca. Suponha uma árvore de valores numéricos, quando vamos realizar a inserção de um número na árvore, pode adotar a seguinte lógica.

* Caso o valor a ser inserido seja menor do que o seu nó pai, inserimos o dado na subárvore esquerda.
* Caso o valor a ser inserido seja maior do que o seu nó pai, inserimos o dado na subárvore direta.

Esse tipo de organização dos dados resulta em um tipo particular de árvore binária, conhecida como **árvore binária de busca**, ou, ainda, ***Binary Search Tree* (BST)**. Veja uma representação dela a seguir.

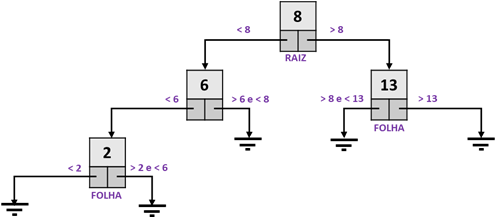
Figura 6 – BST numérica



Note que, no exemplo da Figura 6, a raiz corresponde ao valor 8. Portanto, para a esquerda de 8, só podemos ter valores menores do que ele, e à direita, maiores. Assim, colocamos 13 à direta e 6 à esquerda. O valor 2 é colocado ainda à esquerda de 6, pois é menor do que ele.

Quando precisamos inserir um novo dado dentro da árvore, a inserção precisa ocorrer na posição certa. A Figura 7 apresenta os intervalos em que podemos inserir os dados numéricos.

Figura 7 – BST numérica com informação de inserção

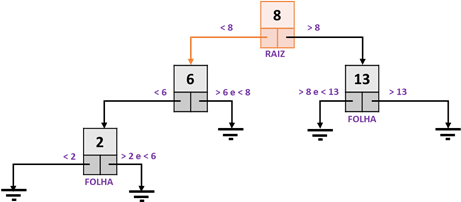


**2.1 INSERÇÃO NA ÁRVORE BINÁRIA DE BUSCA**

Imagine que temos que inserir o valor 7 na BST apresentada na Figura 7. A árvore só tem armazenado no seu programa a posição da raiz, e todos os outros elementos são obtidos a partir delas e seus ponteiros. Portanto, precisamos localizar o local correto para inserir o valor 7.

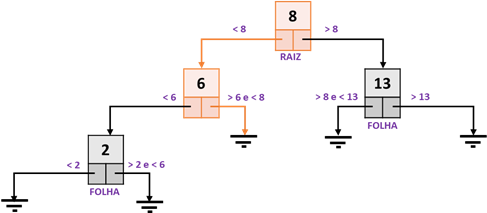
Iniciamos a procura pelo local correto de inserção pela raiz, verificando se o 7 é maior ou menor que o valor posicionado na raiz (valor 8). Como 7 < 8, seguimos pelo ramo esquerdo da árvore, conforme Figura 8.

Figura 8 – Comparação do valor 7 com a raiz 8



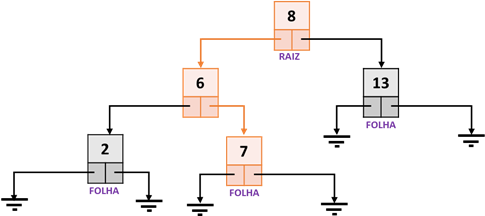
Em seguida, precisamos comparar o valor 7 com o valor 6. Como 7 é maior do que 6, seguimos para o ramo direito do 6. Veja na Figura 9.

Figura 9 – Comparação do valor 7 com o valor 6

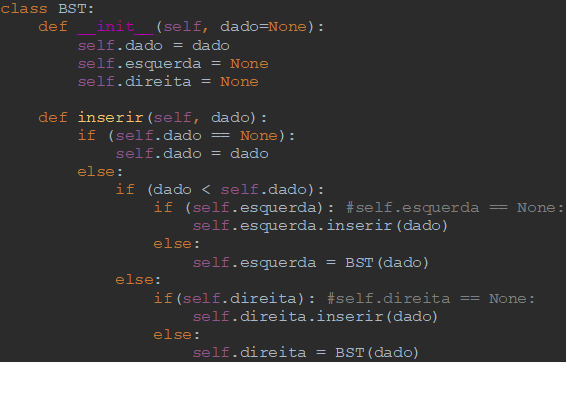


Para a direita de 6 não existe nada. Temos um elemento nulo/vazio. Isso significa que podemos inserir ali. A Figura 10 apresenta a nova BST com o valor inserido na posição correspondente.

Figura 10 – Inserção do valor 7 na BST

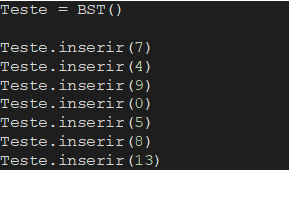


Vamos agora implementar uma BST em linguagem Python. Para isso, criaremos uma classe que irá representar a árvore binária. Se observar as Figuras 6-10, verá que cada elemento da árvore foi representado com dois ponteiros. Aqui iremos chamar os ponteiros de ponteiro esquerdo e ponteiro direito, representando os ramos. Veja o código a seguir.



No constructo de inicialização (\_\_init\_\_), definimos os valores inicias de cada elemento da árvore. A seguir, criamos o método *inserir*, para realizar a inserção de um dado na árvore binária. A inserção em uma árvore binária só pode ocorrer onde existem ponteiros vazios, ou seja, somente nos ramos finais dessa árvore. Não é permitido inserir no meio de uma BST.

O método de inserção realiza esse procedimento de maneira recursiva. Isto é, o método verifica se a posição a qual ele está é passível de inserção (está vazia). Caso não esteja vazia, a função vai sendo chamada recursivamente até que seja possível inserir.



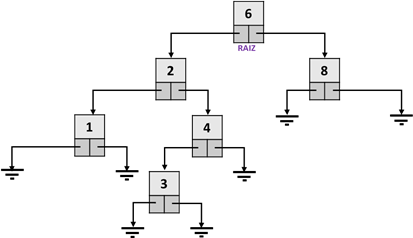
**TEMA 3 – PERCURSO EM BST**

Como percorremos uma BST se não existe um só caminho? Devemos iniciar na raiz sempre? Depende do que deseja e da sua aplicação. Uma árvore binária, por não ser uma estrutura linear, apresenta distintas maneiras de se percorrer por ela para visualizar, manipular ou processar os dados da árvore. Vamos investigar todas essas maneiras agora.

**3.1 PERCURSO EM LARGURA**

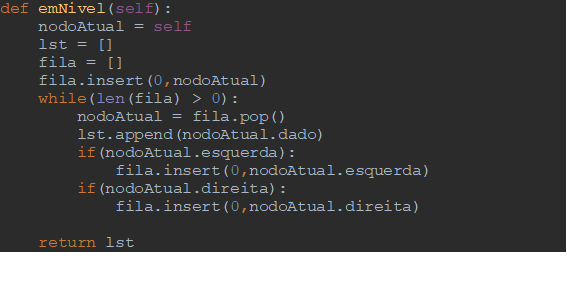
O percurso em largura na árvore binária, também conhecido como percurso em nível, ou *level order*, em inglês.

Figura 11 – Exemplo de árvore binária para percurso



Essa varredura na árvore funciona da seguinte maneira: na Figura 11, é vista uma árvore. O percurso em nível obriga que a árvore passe por todos os elementos de um mesmo nível, para depois passar para os elementos do próximo nível.

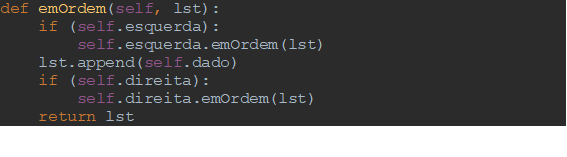
O percurso sempre ocorre de cima para baixo (nível 0 em diante) e da esquerda para a direita. Isso significa que, em nosso exemplo, o percurso iniciaria imprimindo na tela (ou manipulando) a raiz, pois é o único dado de nível 0. Em seguida, iria para o nível 1, da esquerda para a direta (2 e 8). Após, nível 2, da esquerda para a direita (1 e 4). E, por fim, nível 3, com o valor 3. Se a proposta for imprimir os dados na tela, ficaremos com a respectiva sequência impressa: 6, 2, 8, 1, 4, 3.



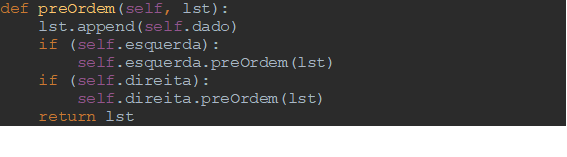
**3.2 PERCURSO EM PROFUNDIDADE**

A segunda maneira de percorrer uma árvore binaria é em profundidade. Neste caso, a árvore não é percorrida horizontalmente como no percurso em largura, mas, sim, é definido um caminho (galho da árvore) e vamos até o fundo dele antes de retornarmos para escolher outro. Existem três formas distintas de percorrer em profundidade. As possibilidades são as que se seguem.

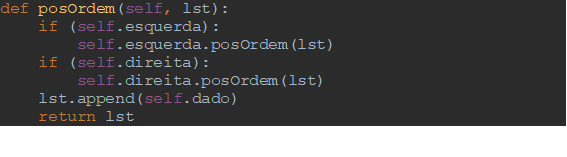
* Percorrer em ordem: listamos os elementos iniciando pelos elementos da esquerda, depois a raiz e, por último, os elementos da direita. Desta forma, os elementos listados ficarão apresentados ordenados de forma crescente.



* Percorrer em pré-ordem: listamos os elementos iniciando pela raiz, depois listamos os elementos da esquerda, e, por fim, os elementos da direita.



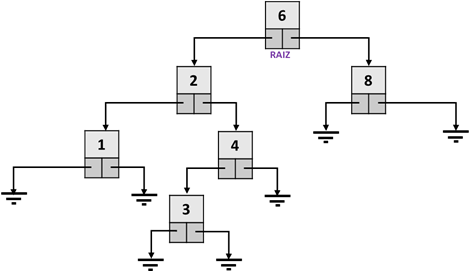
* Percorrer em pós-ordem: listamos os elementos iniciando pelos elementos da esquerda, depois os da direita e, por último, a raiz.



Vamos analisar as listagens possíveis, observando a Figura 12. Para este exemplo, a ordem em que os elementos seriam listados é o que se segue.

* Consultar em ordem: 1, 2, 3, 4, 6, 8.
* Consultar em pré-ordem: 6, 2, 1, 4, 3, 8.
* Consultar em pós-ordem: 1, 3, 4, 2, 8, 6.

Figura 12 – Exemplo de árvore binária para consulta/listagem

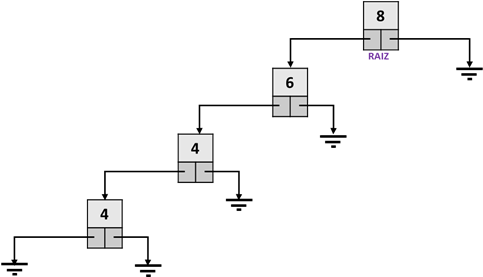


**TEMA 4 – ÁRVORE DE ADELSON-VELSKII E LANDIS (AVL)**

Uma árvore de Adelson-Velskii e Landis, também conhecida como árvore AVL, é uma **árvore binária balanceada**. Mas, afinal, o que é uma árvore balanceada?

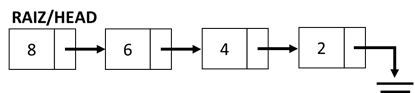
Bom, primeiro precisamos compreender o que é uma árvore desbalanceada. Observe a árvore binária da Figura 13. Note que essa árvore contém unicamente um ramo que segue para o lado esquerdo. É uma árvore sem ramificações.

Figura 13 – Exemplo de árvore binária desbalanceada



Em uma árvore binária convencional, a medida com que vamos tendo muitas inserções de dados, podemos começar a ter nessa árvore algumas ramificações que se estendem muito em altura, gerando piora no desempenho do algoritmo de busca. Veja que, no exemplo anterior, a árvore criada ficou linear, e isso significa que ela irá operar exatamente como uma lista encadeada, afinal, poderíamos reescrever a árvore binária anterior como a lista encadeada apresentada na Figura 14.

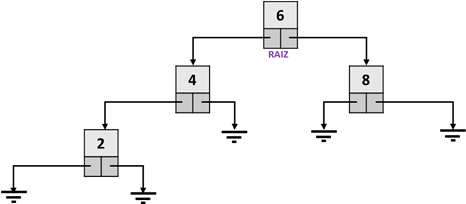
Figura 14 – Árvore binária desbalanceada, com um só ramo, reescrita como uma lista encadeada



Ramificações muito longas acabam dificultando o sistema de busca em uma árvore, piorando o desempenho do algoritmo em tempo de execução. **A árvore AVL tem como objetivo melhorar o desempenho de busca/varredura, balanceando uma árvore binária, evitando ramos longos e gerando o maior número de ramificações binárias possíveis.**

A mesma árvore binária apresentada poderia ter seus elementos rearranjados para que tenha o maior número possível de ramificações. A solução balanceada está apresentada na Figura 15. Note que geramos algumas ramificações que irá facilitar a busca nessa árvore.

Figura 15 – Árvore binária balanceada



**4.1 4.1 CARACTERÍSTICAS DA ÁRVORE AVL**

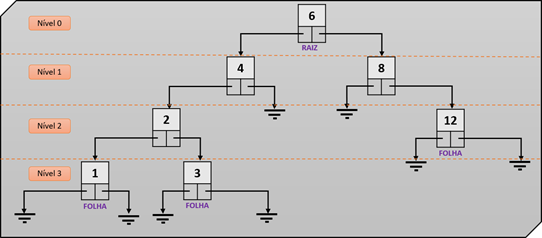
**Uma árvore AVL é uma árvore binária de busca balanceada**. Ou seja, ela contém exatamente as mesmas características de uma BST, todavia, com uma propriedade a mais: **para cada nó da árvore, a diferença de altura entre a subárvore da direta e da subárvore da esquerda será sempre, no máximo, um**. Isso significa que as subárvores daquele nó sempre terão a mesma altura (diferença zero), ou existirá uma diferença de altura entre as subárvores, mas ela é de, no máximo, um (em módulo).

O balanceamento de um elemento da árvore é verificado da seguinte maneira.

* Passo 1: calcula-se a altura relativa daquele elemento para o lado direito da árvore. Neste caso, pegamos o nível mais alto do lado direito daquele elemento e subtraímos do nível do elemento desejado.
* Passo 2: calcula-se a altura relativa daquele elemento para o lado esquerdo da árvore. Neste caso, pegamos o nível mais alto do lado esquerdo daquele elemento e subtraímos do nível do elemento desejado.
* Passo 3: tendo a altura direita e a esquerda calculada, fazemos a diferença entre elas (direta menos esquerda, sempre). Se o cálculo resultar em 2 ou -2, existe um desbalanceamento e uma rotação será necessária na árvore.

Temos, na Figura 16, um exemplo de árvore não balanceada.

Figura 16 – Exemplo de árvore binária não balanceada



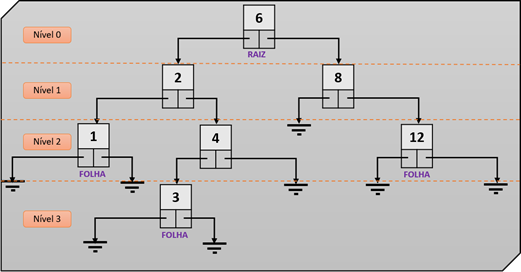
Vamos calcular o balanceamento dessa árvore da Figura 16 para cada elemento. Em destaque, foi colocado o elemento desbalanceado e que precisará ser corrigido.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Elemento** | **Altura Direita** | **Altura Esquerda** | **Direita -Esquerda** | **Balanceado?** |
| **6** | 2 – 0 – 2 | 3 – 0 = 3 | 2 – 3 = -1 | Sim |
| **4** | **0** | **3 – 1 = 2** | **0 – 2 = -2** | **Não** |
| **8** | 2 – 1 = 1 | 0 | 1 – 0 = 1 | Sim |
| **12** | 0 | 0 | 0 | Sim |
| **2** | 3 – 2 = 1 | 3 – 2 = 1 | 1 – 1 = 0 | Sim |
| **1** | 0 | 0 | 0 | Sim |
| **3** | 0 | 0 | 0 | Sim |

O elemento 4 não está balanceado, pois a diferença de altura entre o lado direito e o esquerdo resultou em -2. Para um balanceamento, é obrigatório que essa diferença seja -1, 0 e 1.

Podemos reescrever a árvore anterior de uma maneira diferente, rearranjando os elementos não balanceados de uma maneira que as diferenças de níveis resultem em -1, 0 ou 1. A Figura 17 ilustra uma árvore binária com os mesmos elementos da Figura 16, porém, agora balanceada com o algoritmo AVL.

Figura 17 – Exemplo de árvore binária balanceada (árvore AVL)



Vamos recalcular o balanceamento dessa árvore para cada elemento:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Elemento** | **Altura**  **Direita** | **Altura Esquerda** | **Direita –**  **Esquerda** | **Balanceado?** |
| **6** | 2 – 0 – 2 | 3 – 0 = 3 | 2 – 3 = -1 | Sim |
| **2** | 3 – 1 = 2 | 2 – 1 = 1 | 2 – 1 = 1 | Sim |
| **8** | 2 – 1 = 1 | 0 | 1 – 0 = 1 | Sim |
| **12** | 0 | 0 | 0 | Sim |
| **1** | 0 | 0 | 0 | Sim |
| **4** | 0 | 3 – 2 = 1 | 0 – 1 = -1 | Sim |
| **3** | 0 | 0 | 0 | Sim |

**TEMA 5 – ROTAÇÕES DA ÁRVORE AVL**

Vimos a diferença entre uma árvore não balanceada (Figura 16) e uma árvore balanceada AVL (Figura 17). Porém, como partimos de uma árvore não balanceada e geramos o balanceamento?

Para isso, precisamos realizar rotações em nossa árvore para balanceá-la. Na tabela a seguir, temos o procedimento de rotação que precisa ser realizado, conforme apresentado em (Ascencio, 2011).

|  |  |  |
| --- | --- | --- |
| **Diferença de altura de um nó** | **Diferença de altura do nó filho do nó desbalanceado** | **Tipo de rotação** |
| **2** | 1 | Simples à esquerda |
| **2** | 0 | Simples à esquerda |
| **2** | -1 | Dupla com filho para a direita e pai para a esquerda |
| **-2** | 1 | Dupla com filho para a esquerda e pai para a direita |
| **-2** | 0 | Simples à direita |
| **-2** | -1 | Simples à direita |

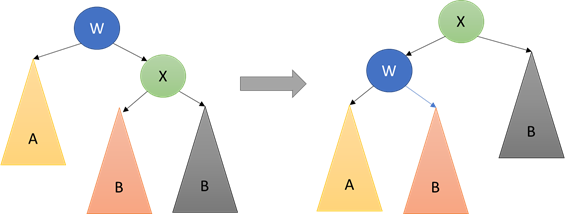
Vamos acompanhar nosso desbalanceamento da Figura 16. O elemento 4 está com desbalanceamento de -2. O nó filho do nó 4, que é o nó 2, está balanceado com valor 0. Deste modo, segundo a tabela de rotações AVL, devemos fazer uma rotação simples à direta.

Essa rotação implicará em colocar o nó 2 no lugar do nó 4. E todos os elementos a seguir dele são rearranjados. O resultado final dessa rotação já foi apresentado na Figura 17. Veremos, a seguir, em mais detalhes como são realizadas as rotações da tabela.

**5.1 ROTAÇÃO SIMPLES À ESQUERDA**

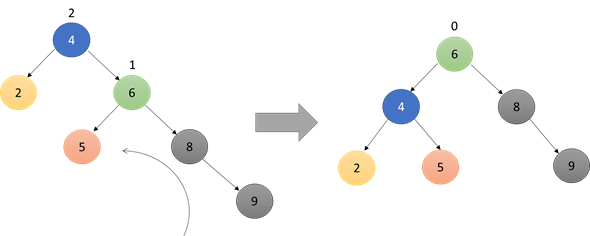
Na rotação simples à esquerda, iremos rotacionar diversos elementos no sentido anti-horário. Veja de maneira genérica na Figura 18, no exemplo, a raiz é W. Com a rotação, levamos ela para a esquerda, deixando de ser raiz. A nova raiz agora será o elemento X, que antes era filho à direta de W.

Figura 18 – Rotação simples à esquerda



A Figura 19 ilustra uma rotação à esquerda com uma BST. Foram empregadas as mesmas cores da Figura 18 para facilitar a identificação dos elementos. Antes da rotação, tínhamos a raiz desbalanceada (balanço 2). E o filho para a direita com balanceamento 1. Isso resulta em uma rotação simples à esquerda.

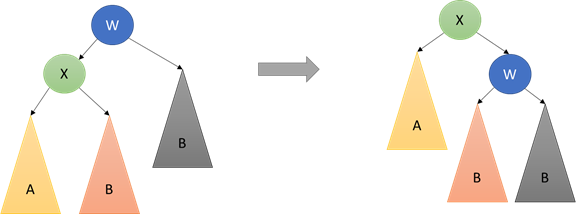
Figura 19 – Rotação simples à esquerda. Exemplo prático



**5.2 ROTAÇÃO SIMPLES À DIREITA**

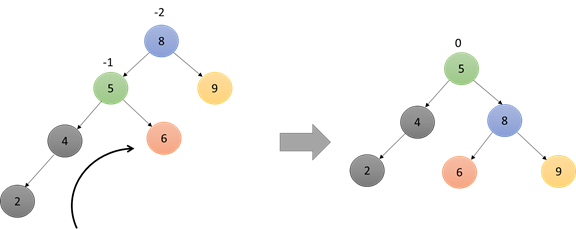
Na rotação simples à direita, iremos rotacionar diversos elementos no sentido horário. Veja de maneira genérica na Figura 20. No exemplo, a raiz é W. Com a rotação, levamos ela para a direita, deixando de ser raiz. A nova raiz agora será o elemento X, que antes era filho à esquerda de W.

Figura 20 – Rotação simples à direita



A Figura 21 ilustra uma rotação à esquerda com uma BST. Foram empregadas as mesmas cores da Figura 19 para facilitar a identificação dos elementos. Antes da rotação, tínhamos a raiz desbalanceada (balanço -2). E o filho para a direita com balanceamento -1. Isso resulta em uma rotação simples à direita.

Figura 21 – Rotação simples à direita. Exemplo prático



**FINALIZANDO**

Ao longo desta etapa, estudamos a estrutura de árvore. Árvores são estruturas de dados não lineares e que são construídas com base em criar ramificações. Por esse motivo, são excelentes no emprego de sistemas de busca.

Uma árvore própria para busca é também chamada de árvore binária de busca (*Binary Serach Tree* – BST), e seus elementos são organizados de tal maneira que facilita a busca.

Todavia, uma árvore, quando apresenta uma inserção de dados não organizada, poderá apresentar ramos muito longos, gerando desbalanceamento na árvore. Podemos balancear a árvore com um algoritmo de balanceamento. Um bastante comum resulta no que chamamos de árvore de Adelson-Velskii e Landis (AVL), também chamado de árvore binária balaceada.

A seguir, temos uma tabela final comparativa do Big-O dos três tipos de árvores investigadas nesta etapa e suas manipulações.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Acesso ao dado** | **Busca** | **Inserção** | **Remoção** |
| **Árvore Binária** | O(n) | O(n) | O(n) | O(n) |
| **BST** | O(n) | O(n) | O(n) | O(n) |
| **AVL** | O(logn) | O(logn) | O(logn) | O(logn) |

**REFERÊNCIAS**

ASCENCIO, A. F. G.; ARAÚJO, G. S. de. **Estruturas de Dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Halt 3, 2010.

BHARGAVA, A. Y. **Entendendo Algoritmos**. Novatec, 2017.

DROZDEK, A. **Estrutura de Dados e Algoritmos em C++**. 4. ed. Cengage Learning Brasil, 2018.

FERRARI, R. et al. **Estruturas de Dados com Jogos.**Elsevier, 2014.

KOFFMAN, E. B.; WOLFGANG, P. A. T. Objetos, Abstração, Estrutura de Dados e Projeto Usando C++. **Grupo GEN**, 2008.

**PROGRAMAÇÃO III**

AULA 5

Prof. Vinicius Pozzobon Borin

**CONVERSA INICIAL**

O objetivo aqui é apresentar os conceitos que envolvem a estrutura de dados do tipo hash. Ao longo da etapa, essa estrutura de dados será conceituada e o seu funcionamento e aplicabilidade apresentados. Investigaremos também o que são funções hash e veremos algumas das mais comuns. Ainda, as possibilidades de implementação da tabela hash com tratamento para colisões serão estudadas. Por fim, veremos a implementação com endereçamento aberto (linear e quadrática) e a implementação com endereçamento em cadeia.

**TEMA 1 – TABELAS HASH**

Tabelas hash, ou somente hash, são estruturas de dados que usufruem de características de arrays, mas que melhoram – e muito – o tempo de inserção e busca dos dados. Para uma melhor compreensão, vejamos o caso a seguir.

**1.1 O PROBLEMA DA QUITANDA DO SEU ZÉ**

Imagine que você trabalha em uma quitanda que vende frutas, verduras, legumes, laticínios, carnes e diversos outros produtos alimentícios. A quitanda do Seu Zé, infelizmente, não acompanhou o avanço tecnológico e até os dias atuais anota todos os preços de produtos vendidos em um caderninho. Para cada produto vendido, é necessário olhar o preço dele no caderno.

Portanto, qual a melhor maneira de se buscar um preço de produto nesse caderno? Bom, caso os produtos estejam fora de ordem, a única maneira seria olhar linha por linha do caderno, equivalente a fazer uma busca sequencial no conjunto de dados. Caso, por sorte, os produtos tenham sido anotados em ordem alfabética, podemos aplicar uma pesquisa binária, aprimorando o desempenho de O(n) para O(logn).

Apesar de conhecermos o bom desempenho de uma busca binária, não queremos deixar o cliente esperando, certo? Imagine ter que aplicar uma busca, seja ela qual for, em um caderno com dezenas ou centenas de linhas. É difícil e inconveniente. Pensando nesse problema é que Seu Zé resolveu contratar a Joana. A moça é conhecida no bairro como “a menina que tudo lembra”. Basicamente, Joana é muito boa em decorar números e palavras e foi capaz de decorar toda a lista de preços da quitanda.

Sendo assim, sempre que você precisa do preço de um produto, em vez de buscar no caderninho, você simplesmente pergunta a Joana, que instantaneamente responde.

O uso de Joana para informar os preços faz com que não tenhamos dependência alguma do tamanho do conjunto de dados. Não importa quantos dados de produtos temos no caderno, ela sempre vai responder instantaneamente. Sendo assim, podemos fazer um cálculo comparativo entre as três abordagens, assumindo que cada execução levaria 100 ms (0,1 s) para ocorrer.

Tabela 1 – Cálculo comparativo

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Busca**  **sequencial** | **Busca**  **binária** | **Busca**  **pela Joana** |
| **Itens no caderno** | O(n) | O(logn) | O(1) |
| **100** | 10 s | 1 s | Instantâneo (0,1 s) |
| **1000** | 1.6 min | 1 s | Instantâneo (0,1 s) |
| **10000** | 16.6 min | 2 s | Instantâneo (0,1 s) |

Bom, a ideia é fantástica, certo? Mas e agora, como criamos e implementamos uma “Joana” em uma estrutura de dados no programa? Podemos juntar as características e vantagens dos arrays, que envolvem acesso instantâneo ao dado em qualquer posição, fazendo também uma busca instantânea? É aqui que entra a construção de um array com base em uma função hash.

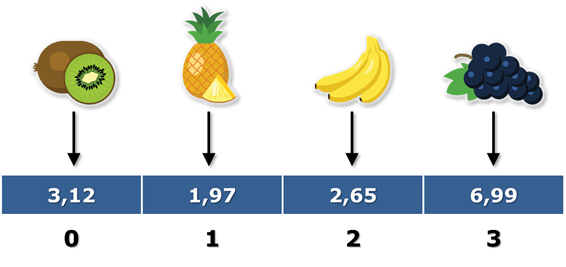
**1.2 FUNÇÃO HASH**

Como você povoaria um array? A maneira mais simples certamente é a de localizar a primeira posição livre e inserir nela um dado. Você sempre inseriu em arrays dessa maneira. Mas a partir de agora, vai aprender a inserir em uma posição qualquer empregando uma função hash.

Uma função hash é uma expressão matemática e/ou lógica que é capaz de receber como entrada um dado, seja ele um número, um caractere, uma string etc., e retornar como resultado um número. Chamamos esse processo de mapeamento. Em nosso exemplo da quitanda, podemos assumir que queremos mapear nomes de produtos (strings) em números.

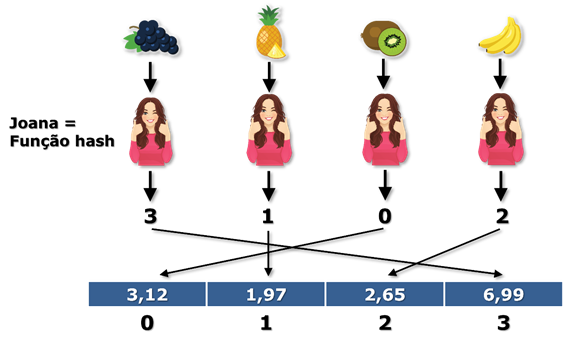
Ou seja, sempre que uma palavra específica aparecer, como uva ou banana, uma função hash precisará convertê-la para um número. Esse número corresponderá a um índice (posição) em uma array. Dentro daquela posição do array calculada, teremos a informação que de fato nos interessa, como o preço daquele produto. A figura 1 mostra um exemplo com um array de quatro elementos, no qual cada índice corresponde a um produto.

Figura 1 – Exemplo com nomes de frutas mapeando números em um array

Crédito: Inspiring/Shutterstock.

Em nosso exemplo da quitanda, a Joana seria nossa função hash. De alguma maneira, ela é capaz de calcular e saber de cabeça qual o valor de cada produto instantaneamente. A figura 2 mostra que uma fruta é mapeada sempre em um número. O número resultante para aquele nome de fruta (string) será sempre o mesmo. Assim, a banana sempre vai retornar 2, por exemplo. A partir do número 2, acessamos o índice do array que contém o preço da banana.

Figura 2 – Mapeamento de produto para um valor numérico

Crédito: Inspiring/Shutterstock; Volha Hlinskaya/Shutterstock.

Pela figura 2, é possível notar que a ordem com que os dados são mapeados no array não é sequencial. Dependendo de quais dados existirem e também da função hash escolhida, o mapeamento pode mudar.

**1.3 A VANTAGEM DO EMPREGO DE UMA FUNÇÃO HASH**

Quando trabalhamos com uma estrutura de dados do tipo array, seja ele estático ou dinâmico, sabemos que, devido a sua **alocação sequencial de dados, somos capazes de acessar qualquer dado dentro dele com o mesmo tempo. Porém, o tempo de busca de um dado em um array é sempre dependente de algum algoritmo**. Portanto:

* Acesso ao dado na memória do array – O(1);
* Busca de um dado no array – depende do algoritmo de busca, pesquisa sequencial O(n) e pesquisa binária O(logn).

Todavia, toda essa análise é feita considerando que estamos cadastrando os dados no array de uma maneira convencional, ou seja, inserimos o dado sempre na próxima posição livre, ordenada ou desordenadamente.

**A função hash tem como objetivo calcular uma posição para inserirmos os dados por meio de um cálculo aritmético e/ou lógico**. Portanto, sempre que a palavra “banana” aparecer, será representada pelo número 2. Sendo assim, sempre que precisamos reaver os dados contidos em “banana”, podemos novamente aplicar o cálculo e, instantaneamente, acessar aquele índice. Dessa maneira, **deixamos de precisar fazer uma varredura no array e tornamos o acesso ao dado independente do tamanho do conjunto de dados**. Por isso, o uso de uma tabela hash nos proporciona:

* Acesso ao dado na memória na tabela – O(1), pois é implementada como um array sequencial;
* Busca de um dado na tabela hash – O(1), pois acessa o índice por meio de uma função hash.

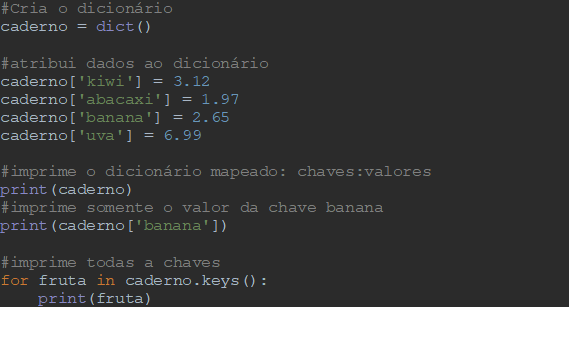
**1.4 A TABELA HASH E SUAS IMPLEMENTAÇÕES**

Um array, quando manipulado com inserção e acesso aos dados por meio do emprego de funções hash, é chamado **tabela hash**.

Os dados que utilizamos como entrada na tabela são chamados **palavras-chave**, ou somente **chaves** (do inglês, ***keys***). Em nosso exemplo, as chaves são os nomes das frutas, que são do tipo string.

É válido ressaltar aqui que algumas linguagens de programação já apresentam estruturas de hash implementadas e prontas para uso. Você provavelmente já trabalhou com estruturas de hash existentes em linguagens de programação. A linguagem C++, por exemplo, trabalha com um contêiner associativo chamado *map*([<https://cplusplus.com/reference/map/map/>](https://cplusplus.com/reference/map/map/)). Já na linguagem Java temos a classe *HashMap*, que é bastante similar a nossa tabela *hash*. Por fim, talvez a estrutura hash mais conhecida por você seja a de **dicionário (*dict*) em linguagem Python.** Podemos criar um dicionário em Python e manipulá-lo conforme segue.

Figura 3 – Dicionário



Note que, para acessar as chaves, utilizamos o método *keys*. Para um estudo aprofundado, recomendamos a leitura do material a seguir, sobre a função hash implementada no Python: [<https://peps.python.org/pep-0456/#conclusion>](https://peps.python.org/pep-0456/#conclusion) (acessado em: 19 set. 2022). Posto isso, é importante que você observe que, ao longo desta etapa, **não vamos trabalhar com estruturas de hash prontas na linguagem de programação, mas sim aprender a construir uma do zero**.

**1.5 APLICAÇÕES DE HASH**

Acerca da aplicabilidade da estrutura de dados do tipo *hash,*a gama de aplicações é bastante grande.

* Podemos manter um rastreamento de jogadas efetuadas por jogadores em jogos como xadrez, damas ou diversos outros jogos com alta quantidade de possibilidades.
* Compiladores necessitam manter uma tabela com variáveis mapeadas na memória do programa. O uso de hashs é muito empregado para tal fim.
* Aplicações voltadas para segurança, como criptografia e autenticação de mensagens e assinatura digital, empregam hashs.
* A estrutura de dados base que permite que as populares criptomoedas – como *bitcoin* – operem são hashs. Elas trabalham com cadeias de hashs altamente complexas para manipular transações, oferecendo segurança e descentralizando as operações.

**TEMA 2 – FUNÇÕES HASH**

A função hash é parte fundamental do processo de criação de uma tabela hash. Uma função hash, também chamada **algoritmo de hash**, **é uma expressão aritmética e/ou lógica específica para resolver uma determinada aplicação**. A função hash não apresenta uma fórmula definida e **deve ser projetada levando-se em consideração o tamanho do conjunto de dados, seu comportamento e os tipos de dados-chave utilizados**.

As funções hash são o cerne na construção das tabelas hash, e o desenvolvimento de uma boa função de hash é essencial para que o armazenamento dos dados, a busca e o tratamento de colisões (assunto abordado ainda nesta etapa) ocorram de forma mais eficiente possível.

Na figura 4, vemos, de maneira genérica, o processo de hash.

Figura 4 – Processo de *hash*



O processo de *hash*é o ato de definirmos um conjunto de chaves como dados de entrada e aplicarmos a eles uma função hash, gerando, na saída, uma posição que será utilizada para acessar um array.

Antes de entrarmos em exemplos de funções hash, é importante ressaltar que, como a possibilidade de funções hash é muito grande, precisamos entender quando elas são consideradas boas funções. Uma boa função hash, em suma, deve ser o que se segue.

* Fácil de ser calculada. De nada valeria termos uma função com cálculos tão complexos e lentos que fizesse com que todo o tempo ganho no acesso à informação com complexidade O(1) fosse perdido calculando uma custosa função de hash.
* Capaz de distribuir palavras-chave o mais uniformemente possível dentro da estrutura do array.
* Capaz de minimizar colisões. Os dados devem ser inseridos de uma forma que as colisões sejam as mínimas possíveis, reduzindo o tempo gasto resolvendo colisões e também reavendo os dados.
* Capaz de resolver qualquer colisão que ocorrer.

**2.1 O MÉTODO DA DIVISÃO**

Talvez a função hashmais comum aplicada é o método da divisão. Nela, a função consiste em convertermos nossas chaves em um valor numérico e dividirmos esse valor por outro, e o resto dessa divisão será o índice do array a ser acessado.

Na equação 1, temos a equação que descreve esse método da divisão. Tal método é bastante rápido em termos computacionais, uma vez que requer unicamente uma divisão.

|  |  |
| --- | --- |
|  | (1) |

Em que k é uma chave qualquer, n é o tamanho do array, e MOD representa o resto de uma divisão**. Quando nossas palavras-chave são valores inteiros, dividimos o número pelo tamanho do array e usamos o resto dessa divisão como sendo a posição a ser manipulada na tabela hash**. Caso uma palavra-chave adotada seja um conjunto grande de valores inteiros (como um número telefônico ou um CPF, por exemplo), poderíamos somar esses valores agrupados (em pares ou trios) e também dividi-los pelo tamanho do vetor, obtendo o resto da divisão.

Exemplificando, um número telefônico com 8 valores, como 9988-2233, pode ser quebrado em pares e somado para gerar um só valor: . Esse valor é usado no método da divisão.

Para um array de tamanho 12 (n) e uma chave de valor 242 (), o resultado da função será , ou seja, adotaríamos o índice 2 para manipular essa chave na tabela. Existem alguns arrays que devem ser minuciosamente escolhidos para não gerar povoamentos de tabelas hashruins. Por exemplo, utilizar 2 ou múltiplos de 2 para o valor de *n*tende a não ser uma boa escolha. Isso porque o resto da divisão por dois ou qualquer múltiplo seu sempre resultará em um dos bits menos significativos, gerando um número bastante elevado de colisões de chaves.

Em hash, precisamos trabalhar com números naturais para definir as posições na tabela, uma vez que linguagens de programação indexam as estruturas de dados usando esse tipo de dado numérico. Desse modo, precisamos que nossas chaves sejam também valores naturais. Caso não sejam, precisamos encontrar uma forma de transformá-las para tal.

Para chaves com caracteres alfanuméricos, podemos adotar a mesma equação 1, fazendo pequenas adaptações. Convertemos os caracteres para números decimais seguindo uma codificação (tabela ASCII, por exemplo), somamos os valores, dividimos pelo tamanho do vetor e obtemos o resto da divisão como posição da palavra-chave na tabela hash(equação 2).

|  |  |
| --- | --- |
|  | (2) |

Adotar números primos, especialmente aqueles não muito próximos aos valores de potência de 2, tende a ser uma boa escolha para o tamanho do vetor com palavras-chave alfanuméricas.

Por exemplo, suponha que temos 2000 conjuntos de caracteres para colocar em uma tabela hash. Definiu-se que realizar uma busca, em média, em até três posições antes de encontrar um espaço vazio é considerado aceitável para essa aplicação. Assim, fazendo para o valor de *n*,podemos adotar um número primo próximo de 666, mas não muito próximo de um múltiplo de 2. Podemos usar o valor 701 para *n*. Nossa função *hash* resultante seria .

Por fim, quando trabalhamos com strings como palavras-chave, devemos tomar cuidado com palavras que contenham as mesmas letras, mas em ordens diferentes (anagrama). Por exemplo, uma palavra-chave com quatro caracteres como *ROMA*poderá gerar o mesmo resultado que uma palavra-chave chamada *AMOR*, pois os caracteres são os mesmos rearranjados de outra maneira. Convertendo ambas as palavras para ASCII, decimal, e somando os valores, obtemos 303. Suponha que *n = 13.*Teríamos:



Uma função de hash deve ser cuidadosamente definida para tratar esse tipo de problema, caso ele venha a ser recorrente. Do contrário, teremos excessivas colisões.

**2.2 A HASH UNIVERSAL**

Considerando que uma chave k qualquer tem igual probabilidade de ser inserida em qualquer uma das posições de um vetor, em um pior cenário, seria possível que um conjunto de chaves a serem inseridas caiam sempre na mesma posição do vetor, caso utilizem a mesma função hash h(k). Portanto, a complexidade para manipular nessa hash será . Podemos minimizar esse tipo de problema escolhendo uma função hash aleatoriamente dentro de um universo H de funções. A essa solução chamamos **hash universal**.

Na hash universal, no início da execução de um algoritmo de inserção, sorteamos aleatoriamente uma função de hash dentro de uma classe de funções cuidadosamente desenvolvida para a aplicação desejada. A aleatoriedade evita que qualquer entrada de dado resulte no pior caso.

É bem verdade que a aleatoriedade poderá nunca resultar em um caso perfeito, no qual nenhuma colisão ocorre, mas teremos sempre uma situação em média aceitável.

Uma classe H de funções de hash é considerada universal se o número de funções for igual a . A probabilidade de que ocorra será com a seleção aleatória. A prova matemática da hash universal pode ser encontrada no livro do Cormen (2011).

**2.2.1 IMPLEMENTAÇÃO MATEMÁTICA DE HASH UNIVERSAL**

Imaginemos um número primo p e um conjunto de valores . Definimos que , ou seja, é o conjunto excluindo o valor zero. Podemos adotar uma classe de funções H que seja dependente desse número primo p e do seu conjunto (equação 3):

|  |  |
| --- | --- |
|  | (3) |

Onde e . A variação das constantes a e b constituem diversas possibilidades para essa classe de funções dependente de um valor primo p. Se assumirmos o número primo , um vetor de dimensão e também e , obtemos pela equação 3 que .

Existe outro método bastante conhecido de implementação de hash universal que emprega matrizes aleatórias. Suponha que você tem um conjunto de dados de entrada de tamanho bits. Você deseja produzir palavras-chave de tamanho bits. Criamos, então, uma matriz binária aleatória de dimensão . A função hash dessa classe será, portanto, .

Você deve considerar seus dados como sendo valores binários. Esses valores podem ser valores inteiros ou mesmo caracteres convertidos para valores binários. Por exemplo, suponha que você tem dados de 4 bits e precisa gerar chaves com 3 bits. Faremos uma matriz . Uma possível matriz binária aleatória seria:

|  |  |
| --- | --- |
|  | (4) |

Assumindo um dado de entrada como sendo 1011, multiplicamos pela matriz aleatória e obtemos um resultado de 3 bits (equação 5). A geração aleatória dessa matriz binária para cada nova chave caracteriza um conjunto de funções H que pode ser considerada uma hash universal, pois a possibilidade de uma matriz gerada ser igual a outra é bastante pequena.

|  |  |
| --- | --- |
|  | (5) |

**TEMA 3 – IMPLEMENTANDO UMA TABELA HASH**

Vamos imaginar que você trabalha para o Instituto Brasileiro de Geografia e Estatística (IBGE) e está implementando um código para catálogo de dados de diferentes estados brasileiros. Entre esses dados, você precisará armazenar muitas informações, como a capital do estado, a população total, a lista de cidades, o PIB do estado, o índice de criminalidade, o nome do governador, entre inúmeros outros dados.

Você vai implementar esse cadastro usando uma tabela hash. Vamos adotar a sigla de cada estado como sendo uma palavra-chave. Como sabemos de antemão que a sigla de todos os estados brasileiros é sempre constituída de dois caracteres alfanuméricos, vamos converter cada caractere para seu valor em decimal seguindo o padrão da tabela ASCII (para consulta, veja [<http://www.asciitable.com/>](http://www.asciitable.com/), acessado em: 19 set. 2022). Então, vamos definir uma função hash como sendo a soma em decimal de ambas as letras e dividir o resultado pelo tamanho de nosso array (dimensão 10). A posição de inserção no vetor será o resto dessa divisão.

Consideraremos ambos os caracteres em letras maiúsculas. Por exemplo, para o estado do Paraná, sigla PR, o caractere ASCII da letra P é 80, e o da letra R é 82. A soma desses valores resulta em 162. Dividindo esse valor pelo tamanho do vetor (dimensão 10) e obtendo somente o resto dessa divisão, temos o valor 2. Esse será, portanto, a posição de inserção dos dados referentes ao estado do Paraná. Assim, mesmo que PR seja o primeiro dado a ser cadastrado no vetor, ele não será posicionado na posição zero do vetor, mas sim na posição dois.

De forma análoga, podemos calcular a posição no vetor do estado do Rio Grande do Sul, sigla RS. Temos R = 82 e S = 83. A soma de ambos os valores será 165, e o resto da divisão por 10 resultará no valor 5, valor esse que corresponde à posição de inserção desse dado no vetor.

Na tabela a seguir, temos o cálculo da posição de inserção para alguns estados brasileiros utilizando a função hash.

Tabela 2 – Cálculo da posição de inserção

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Sigla*** | **ASCII (Dec)** | **ASCII (Dec)** | **Soma** | **Posição**  **(Soma MOD 10)** |
| ***PR*** | P (80) | R (82) | 162 | **2** |
| ***SC*** | S (83) | C (67) | 150 | **0** |
| ***RS*** | R (82) | S (83) | 165 | **5** |
| ***SP*** | S (83) | P (80) | 163 | **3** |
| ***RR*** | R (82) | R (82) | 164 | **4** |
| ***RJ*** | R (82) | J (74) | 156 | **6** |
| ***AL*** | A (65) | L (76) | 141 | **1** |
| ***DF*** | D (68) | F (70) | 138 | **8** |
| ***PE*** | P (80) | E (69) | 149 | **9** |

Na figura a seguir, temos os dados calculados usando a função hash e agora inseridos nas suas respectivas posições. Cada estado resultou em uma posição diferente na tabela hash. Veja a inserção dos dados na tabela.

Figura 5 – Array com dados de estados brasileiros inseridos utilizando uma função hash



É interessante analisar que, alterando o tamanho de nosso vetor, podemos alterar também a posição de inserção de cada valor. Por exemplo, se o vetor fosse de dimensão 12, a sigla PR seria posicionada no índice 6, e não no índice 2, conforme mostrado para um tamanho 10.

**3.1 IMPLEMENTAÇÃO EM PYTHON: FUNÇÃO HASH**

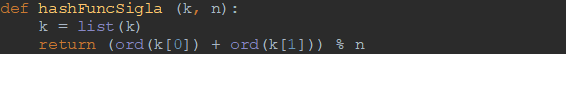
A seguir, temos uma implementação de uma função hash usando o método da divisão e empregando valores numéricos. A função criada é bastante simples. Ela recebe a chave e o tamanho de uma lista em Python e retorna o resto dessa divisão (símbolo de %).

Figura 6 – Implementação em Python



Uma variação dessa mesma função do método da divisão pode ser vista a seguir. O código funciona para dois caracteres, podendo ser aplicado para as siglas de estados brasileiros.

Figura 7 – Implementação em Python – variação

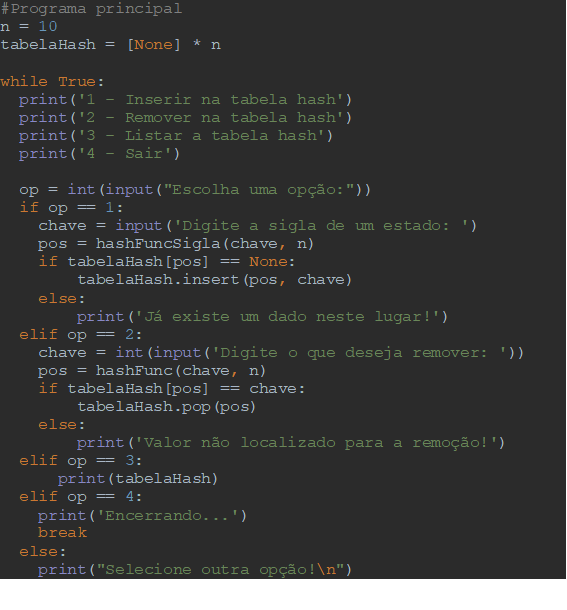


Note que primeiro convertemos a chave recebida, que é uma string, em uma lista. Em seguida, usando uma função do Python chamada ord, convertemos cada caractere em seu respectivo valor decimal da tabela ASCII. Por fim, fazemos a soma e pegamos o resto da divisão.

**3.2 IMPLEMENTAÇÃO EM PYTHON COMPLETA**

O código a seguir implementa um array com inserção e remoção empregando uma função hash. Foi criado um menu de seleção, contendo as opções de inserir, remover e listar a estrutura de dados. Note que, **se houver uma mesma chave que caia em uma posição em que já existe algum dado, o código não permite a inserção do dado. Quando essa situação ocorre, na verdade temos uma colisão**. Vamos aprender a tratar colisões no próximo tópico. Quando aprendermos esse assunto, vamos reformular esse código para atender às colisões também.

Figura 8 – Array com inserção e remoção de função hash



**TEMA 4 – COLISÕES EM TABELAS HASH**

E se duas palavras-chave resultarem em uma mesma posição dentro do array, como procedemos? Quando isso ocorre, dizemos que uma colisão ocorreu. Mas como tratamos essa colisão? Podemos ainda inserir um dado colidido ou precisamos descartá-lo?

É basicamente **impossível escrevermos uma função hash que seja livre de colisões**. Não importa a função hash nem a aplicação, colisões sempre ocorrerão. Podemos sim tratar essas colisões e inserir um dado colidido em outra posição da tabela hash, mas, para isso, vamos precisar investigar e estudar maneiras de tratar esse tipo de problema.

**4.1 COLISÕES COM ENDEREÇAMENTO ABERTO**

A maneira como vamos tratar as colisões depende muito do tipo de endereçamento usado para construir a tabela hash. O primeiro tipo é o **endereçamento aberto**. Nele, **cada posição da estrutura de dados só pode conter uma única palavra-chave**. Isso significa que, caso uma segunda chave seja calculada para entrar em uma posição já preenchida, precisaremos aplicar um algoritmo para inserir em outra posição livre.

**4.1.1 TENTATIVA LINEAR**

Na tentativa linear, sempre que uma colisão ocorre, tenta-se posicionar a nova chave no próximo espaço imediatamente livre do array. Vamos compreender o funcionamento por meio de um exemplo. Queremos preencher um vetor de dimensão 10 com palavras-chave iguais à sigla de cada estado brasileiro (dois caracteres). O cálculo de cada posição é feito utilizando o método de divisão para caracteres alfanuméricos (equação 2).

Agora, imaginemos uma situação inicial em que temos o vetor preenchido com três estados, conforme a tabela com as siglas PR, RS e SC, em que cada sigla está em uma posição distinta do vetor.

Tabela 3 – Siglas em posições distintas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Sigla*** | **ASCII (Dec)** | **ASCII (Dec)** | **Soma** | **Posição**  **(Soma MOD 10)** |
| ***PR*** | P (80) | R (82) | 162 | **2** |
| ***SC*** | S (83) | C (67) | 150 | **0** |
| ***RS*** | R (82) | S (83) | 165 | **5** |

Temos, na figura a seguir, os dados inseridos nas posições 0, 2 e 5. Isso significa que essas três posições estão com o status de ocupado, enquanto todas as outras estão com status livre.

Figura 9 – Array com dados de estados brasileiros inseridos



Agora, queremos inserir o estado do Amazonas (AM) nesse vetor. Utilizando a equação 2, o somatório de seus caracteres em ASCII resulta em 142 (ADEC + MDEC). O resto da divisão pelo tamanho do vetor (10) resultará na posição 2. Essa posição já está ocupada pelo estado do Paraná (PR), **resultando em uma colisão**. Sendo assim, é necessário tratar a colisão e resolvê-la de alguma maneira.

No algoritmo da tentativa linear, quando ocorre essa colisão, segue-se para a **posição subsequentemente livre.** Após a posição 2, seguimos para a posição 3. Essa posição está vazia, de modo que podemos inserir o estado do AM nela. O resultado é visto na figura a seguir.

Figura 10 – Vetor com inserção e colisão por tratativa linear



Continuando, vamos inserir mais um estado nesse array, o estado do Acre (AC). O somatório de seus caracteres em ASCII resulta em 132 (ADEC + CDEC) e, portanto, o resto da divisão pelo tamanho do vetor (10) resultará, mais uma vez, na posição 2.

Conforme visto anteriormente, a posição 2 está ocupada pelo PR. Seguimos para a próxima posição pela tentativa linear, porém a posição 3 também está ocupada, pelo estado do AM. Assim, incrementamos novamente nossa posição e atingimos a posição 4 – vazia – e podemos fazer a inserção nela. A figura a seguir ilustra nosso exemplo.

Figura 11 – Vetor com inserção e colisão por tratativa linear



O algoritmo de tentativa linear pode ficar buscando uma posição vazia indefinidamente até chegar ao final do vetor. Caso não encontre nenhuma posição vazia, ele retorna ao início e busca até voltar à posição inicialmente testada. Caso nenhum local livre seja localizado, a palavra-chave não pode ser inserida.

**4.1.2 TENTATIVA QUADRÁTICA**

Na tentativa quadrática, sempre que uma colisão ocorre, tenta-se posicionar a nova chave no próximo espaço que está a *d* posições de distância do primeiro espaço testado, onde 1≤ d ≤ n, em que *n*é o tamanho do vetor. A função hash adotada como exemplo será novamente o método de divisão para caracteres alfanuméricos (equação 2). Agora, imaginemos um estado inicial em que temos o vetor preenchido com três siglas.

Tabela 4 – Estado com três siglas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Sigla*** | **ASCII (Dec)** | **ASCII (Dec)** | **Soma** | **Posição**  **(Soma MOD 10)** |
| ***PR*** | P (80) | R (82) | 162 | **2** |
| ***SC*** | S (83) | C (67) | 150 | **0** |
| ***RS*** | R (82) | S (83) | 165 | **5** |

Nenhuma das três siglas inseridas na tabela apresentaram colisão, e todas as posições calculadas pela função hash foram diferentes. Vemos essa representação na figura a seguir.

Figura 12 – Vetor com dados de estados brasileiros inseridos



Vamos inserir o estado do Pará (PA) nesse array. O cálculo pelo método da divisão para caracteres alfanuméricos resulta na posição 5 para a sigla PA, posição na qual já temos o estado RS. Vejamos as etapas para a inserção:

1. *Status inicial.*Calcula-se a posição inicial e inicializa-se a variável *i* com o valor 1.
   * 
   * A variável i é inicializada: .
2. *Etapa*1. Verifica-se a posição 5. Como ela está ocupada, incrementa-se o valor de i e realiza-se o cálculo da função hash novamente, resultando na posição 6. Incrementa-se o contador.
   * Posição 5 já está ocupada;
   * 
   * 
3. *Etapa 2.*
   * Posição 6 está livre. Inserção realizada.

Figura 13 – Vetor com dados de estados brasileiros inseridos



Vamos continuar inserindo nesse vetor com o estado do Amapá (AP). O cálculo pelo método da divisão para caracteres alfanuméricos resulta na posição 5, mais uma vez, para a sigla PA.

Vejamos as etapas de inserção.

1. *Status inicial.*Calcula-se a posição inicial e inicializa-se a variável *i* com o valor um.
   * 
   * A variável i é inicializada: .
2. *Etapa 1*. Verifica-se a posição 5. Como ela está ocupada, incrementa-se o valor de i nela e realiza-se o cálculo da função hash novamente, resultando na posição 6. Incrementa-se o contador.
   * Posição 5 já está ocupada;
   * 
   * 
3. *Etapa 2*. Verifica-se a posição 6. Como ela está ocupada, incrementa-se o valor de i nela e realiza-se o cálculo da função hash novamente, resultando na posição 8. Incrementa-se o contador.
   * *Posição 6 já está ocupada.;*
   * 
   * 
4. *Etapa 3.*
   * *Posição 8 está livre. Inserção realizada.*

Figura 14 – Vetor com dados de estados brasileiros inseridos



**4.2 ENDEREÇAMENTO EM CADEIA**

O segundo tipo é o **endereçamento em cadeia**. Nele, **cada posição da estrutura de dados pode contar diversas palavras-chave.** Na prática, para implementá-lo, vamos precisar trabalhar com listas encadeadas.

Nesse cenário, temos também um array de dimensão *n* para representar a tabela. Porém, cada posição do vetor armazenará um endereço para o início de uma lista encadeada. A função hash a ser empregada aqui continua sendo qualquer uma das já apresentadas no decorrer desta caminhada, portanto, continuaremos adotando o método da divisão em nossos exemplos. Vamos ao nosso exemplo dos estados.

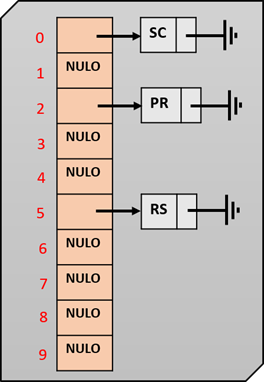
Tabela 5 – Siglas dos estados

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Sigla*** | **ASCII (Dec)** | **ASCII (Dec)** | **Soma** | **Posição**  **(Soma MOD 10)** |
| ***PR*** | P (80) | R (82) | 162 | **2** |
| ***SC*** | S (83) | C (67) | 150 | **0** |
| ***RS*** | R (82) | S (83) | 165 | **5** |

A inserção dos dados agora é tratada da seguinte maneira: cada chave a ser inserida é alocada como um elemento na memória e, em seguida, seu endereço é colocado em uma lista encadeada referente à posição calculada. Por exemplo, a sigla PR, que tem a posição 2 pela tabela, terá seu endereço posicionado nessa posição do vetor.

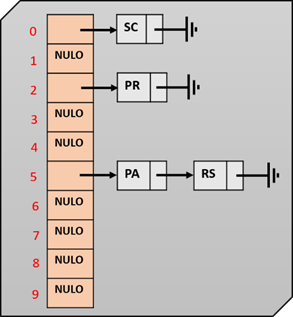
Todas as siglas calculadas na tabela, como estão sozinhas em cada posição, representam o *Head* de uma lista encadeada. Assim, as posições 0, 2 e 5 contêm uma lista encadeada com um só elemento.

Figura 15 – Endereçamento em cadeia com dados de estados brasileiros inseridos da tabela



Vamos inserir a sigla PA nesse vetor da figura usando o endereçamento em cadeia. A sigla PA também corresponderá à posição 5 pelo método da divisão. Nessa posição já temos a sigla RS, resultando em uma colisão.

Figura 16 – Endereçamento em cadeia: adicionando PA à posição 5



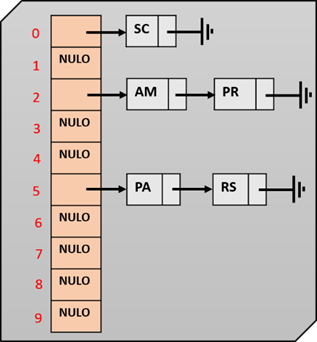
A forma como as colisões são tratadas por esse tipo de endereçamento é diferente. Nele, não é necessário encontrarmos uma nova posição no vetor para alocarmos o elemento colidido. Basta inseri-lo na mesma posição 5 calculada, mas como mais um elemento da lista encadeada simples.

A inserção é dada sempre antes do *Head*(início da lista encadeada). Assim, a sigla PA virará o novo *Head* da lista da posição 5, apontando para a sigla RS que está na segunda posição da lista. Por se tratar de uma lista não circular, o último elemento conterá um ponteiro nulo para o próximo elemento.

De forma análoga, na figura 17, acrescentamos a sigla AM. A posição calculada para ela pelo método da divisão resulta na posição 2 do vetor. O elemento será de fato inserido nessa posição, sem a necessidade de encontrar outra posição.

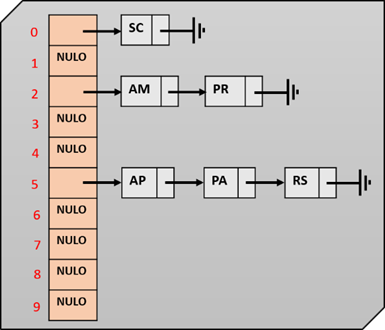
Como já existe a sigla PR na posição 2, insere-se AM na lista encadeada dessa posição. A sigla AM será o *Head* e apontará para PR, que apontará para nulo. Note que sempre que for necessário buscar uma chave desse vetor, basta recalcular a posição usando a função hash e, em seguida, varrer a lista encadeada simples daquela posição até localizar a palavra-chave correspondente.

Figura 17 – Endereçamento em cadeia: adicionando AM à posição 2



Podemos continuar inserindo elementos indefinidamente em cada lista encadeada. Por exemplo, se um terceiro elemento colidir na posição 5, acontecerá o mesmo processo. A sigla AP precisa entrar na lista encadeada. Assim, ela será colocada no lugar do *Head* da lista encadeada da posição 5, deslocando todos os outros elementos dessa lista.

Figura 18 – Endereçamento em cadeia: adicionando AP à posição 5



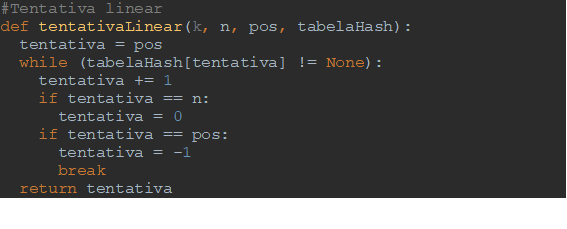
**TEMA 5 – IMPLEMENTANDO COLISÕES E DESEMPENHO HASH**

Vamos implementar uma tabela hash tratando as colisões em linguagem Python.

**5.1 ENDEREÇAMENTO ABERTO E TENTATIVA LINEAR**

Para o endereçamento aberto, implementamos em linguagem Python com listas, que são arrays dinâmicos. A seguir, podemos ver a implementação para a tentativa linear.

Figura 19 – Tentativa linear

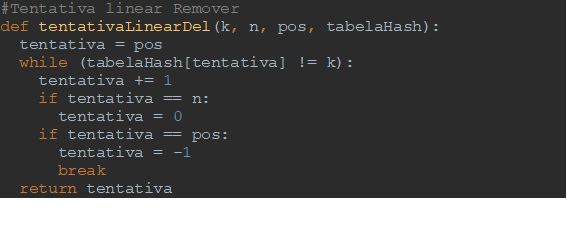


A função implementada recebe como parâmetro a tabela, a posição a ser inserida e também a chave e o tamanho do vetor. Criamos uma variável adicional local chamada *tentativa*que irá receber como valor inicial a posição inicial a ser testada para a inserção. Caso a posição não esteja vazia, precisamos incrementar a posição de tentativa até aparecer uma posição vazia.

Todavia, note que a posição a ser inserida ocasionalmente pode ser a última, ou uma das últimas, do array. Mas ainda temos posições livres no início do array. Portanto, temos uma condicional que verifica se atingimos o final do array e voltamos para o início. Temos também outra condição que identifica se todas as posições já foram testadas, retornando um código de erro.

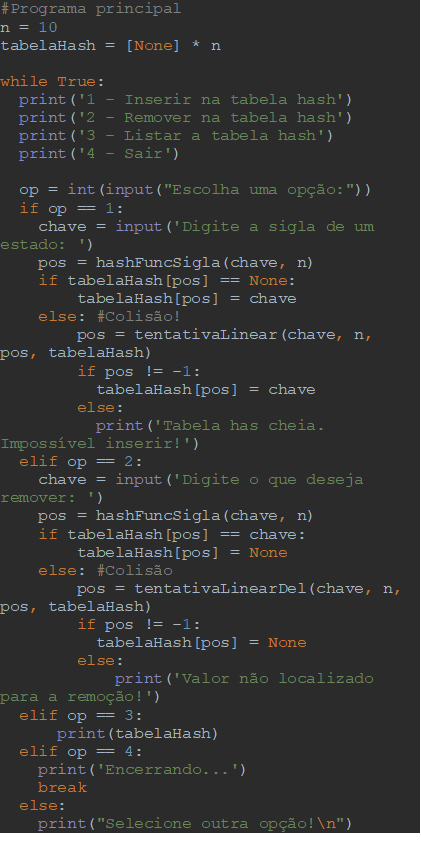
A seguir, vemos também o código para remover um dado com colisões sendo tratadas com tentativa linear.

Figura 20 – Tratamento com tentativa linear



O código completo de manipulação da hash é visto a seguir.

Figura 21 – Código completo



**5.2 FATOR DE CARGA**

Suponha que você precisa armazenar o preço de 100 produtos da quitanda do Seu Zé na tabela hash. Se sua tabela hash tiver 100 espaços no array, na melhor hipótese, sem colisões, cada chave terá seu espaço próprio. Isso significa que o fator de carga desse exemplo será 1. O fator de carga é sempre calculado como sendo:



No exemplo, 100 produtos (chaves) para 100 espaço dá 100 dividido por 100, resultando em 1. Observe a figura a seguir. Temos array de dez espaços. Se retomarmos o exemplo dos estados brasileiros, temos 26 estados no Brasil, mais o Distrito Federal.

Figura 22 – Exemplo para o fator de carga



Para calcular o fator de carga, fazemos: 27/10 = 2,7. O fator de carga está bastante alto! Idealmente, precisamos deixar o fator de carga abaixo de 1, pois assim teremos certeza de que existirão espaços livres para todas as chaves. Sempre que o fator de carga começar a aumentar para uma determinada aplicação, precisaremos ficar atentos e redimensionar o tamanho do conjunto de dados.

**5.3 DESEMPENHO DE HASH**

No pior caso, uma tabela hash tem tempo de execução O(n). Isso ocorre porque o pior caso contempla o fato de sempre existirem colisões e sempre precisarmos tratá-las. Para fins de comparação, estamos apresentando dessa vez não só o pior caso, como também o caso médio das manipulações da tabela hash.

Tabela 6 – Busca, inserção e remoção

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Tabela hash**  **(caso médio)** | **Tabela hash**  **(pior caso)** | **Array** | **Listas encadeadas** |
| **Busca** | O(1) | O(n) | O(1) | O(n) |
| **Inserção** | O(1) | O(n) | O(n) | O(1) |
| **Remoção** | O(1) | O(n) | O(n) | O(1) |

Preste atenção ao caso médio para tabelas hash. As tabelas hash são tão velozes quanto arrays para busca. Elas são tão velozes quanto as listas para inserção e remoção. Ou seja, ela é o melhor entre dois mundos! É claro que, para o pior caso, ela acaba sendo lenta. Portanto, precisamos evitar esse pior caso, evitando ao máximo as colisões.

**FINALIZANDO**

Nesta etapa, aprendemos sobre a estrutura de dados do tipo hash. O objetivo da hashfoi construir uma estrutura de dados capaz de obter tempo de acesso constante às informações contidas nela, independentemente do tamanho do conjunto de dados.

Vimos que tabelas hasharmazenam palavras-chave que servem para acessar dados. Essas palavras-chave são armazenadas em posições de um vetor calculadas por meio de funções hash. Vimos também que essas funções são expressões matemáticas e/ou lógicas e aprendemos duas das mais conhecidas.

Analisamos diferentes algoritmos para resolver os problemas das colisões, ou seja, quando duas chaves precisam ser posicionadas em uma mesma posição. Aprendemos a tentativa linear e a tentativa quadrática para resolver esse problema usando endereçamento aberto.

**REFERÊNCIAS**

CORMEN, T. H. **Algoritmos**. Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

**PROGRAMAÇÃO III**

AULA 6

Prof. Vinicius Pozzobon Borin

**CONVERSA INICIAL**

O objetivo desta etapa é apresentar os conceitos que envolvem a estrutura de dados do tipo grafo. Ao longo deste documento será conceituado essa estrutura de dados, bem como as formas de representação de um grafo. São elas: matriz de adjacências e lista de adjacências.

Ainda, será mostrado como realizar a descoberta de um grafo por intermédio de dois algoritmos distintos: algoritmo de busca por largura; e algoritmo de busca por profundidade.

Por fim, será apresentado um algoritmo bastante tradicional para encontrar o caminho mínimo entre dois vértices em um grafo: algoritmo de Dijkstra.

**TEMA 1 – CONCEITOS DE GRAFOS**

Anteriormente, já investigamos estruturas de dados que funcionam de uma forma linear, como a lista encadeada, e também vimos as não lineares, na forma de árvores binárias. Porém, uma árvore, seja ela binária ou não, ainda segue um aspecto construtivo fixo da sua estrutura por meio da geração de ramos e subdivisões. Nesta etapa, vamos investigar uma estrutura de dados que pode vir a ter sua construção sem padrão de construção nenhum, os grafos.

Para entendermos o conceito e a aplicabilidade de grafos, vamos começar com um exemplo prático. Imaginemos que fomos contratados por uma empresa para mapear ruas, estradas e rodovias de uma cidade. Desse modo, recebemos um mapa aéreo da cidade e, por intermédio de um processamento digital de imagens, extraímos todas as ruas e avenidas dessa cidade, resultando em um mapa destacado semelhante ao da Figura 1, a seguir.

Figura 1 – Mapa rodoviário de uma cidade hipotética

Crédito: ShustrikS/Shutterstock.

Com esse mapa processado, precisaremos realizar o mapeamento das estradas com o objetivo de desenvolver um *software* que calcula as melhores rotas de um ponto de origem até um destino no mapa.

Para realizar esse cálculo de rotas, podemos mapear a rede rodoviária da Figura 1 em uma estrutura de dados do tipo grafo. Assim, podemos transformar cada ponto de intersecção entre ruas e avenidas em um **vértice**de um grafo, e cada conexão entre duas intersecções em uma **aresta** de um grafo.

Na Figura 2, a seguir, há uma ilustração de um exemplo de mapeamento de uma região da cidade de Curitiba, em que os círculos pretos são os vértices de intersecção, e as linhas pretas, as arestas. Embora na referida figura somente algumas ruas estejam mapeadas, poderíamos fazer esse processo para a cidade inteira. Com o mapeamento pronto, bastaria que aplicássemos um algoritmo para encontrar o menor caminho do grafo e encontrar as rotas desejadas de um ponto ao outro.

Figura 2 – Mapa rodoviário de uma cidade hipotética com vértices e arestas do grafo marcadas

Crédito: Vinicius Pozzobon Borin.

**1.1 DEFINIÇÕES DE GRAFOS**

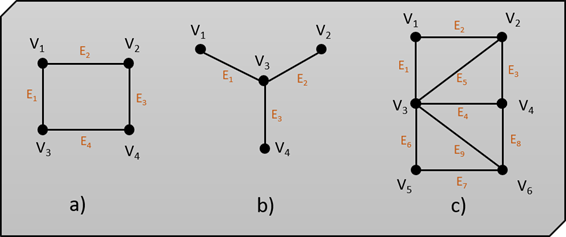
Um **Grafo** Gé um conjunto de vértices conectados por meio de arestas sem uma distribuição fixa ou padronizada.

**Vértices** V de um grafo são seus pontos. Cada ponto poderá ser um ponto de encontro entre caminhos (rotas) de um grafo, ou então esse vértice poderá conter informações relevantes para o grafo, como dados de informações de cadastros. Tudo dependerá da aplicação.

**Arestas**Esão as linhas de conexão entre vértices. Cada aresta conecta dois vértices. Nem todo vértice precisa ter uma aresta conectada, pois pode permanecer isolado caso o grafo assim seja construído.

A Figura 3, a seguir ilustra três exemplos de grafos. No exemplo da Figura 3ª, há 4 vértices e 4 arestas. Na Figura 3b, também há 4 vértices, mas somente 3 arestas. Na Figura 3c, é interessante perceber que um mesmo vértice chega a conter 5 arestas partindo dele. A quantidade de vértices e arestas em um grafo pode ser ilimitada, não havendo restrições.

Figura 3 – Exemplos de grafos

Crédito: Vinicius Pozzobon Borin.

Para a construção do grafo não existe uma regra fixa. Qualquer arranjo de vértices e arestas pode ser considerado um grafo. Vejamos alguns conceitos atrelados a essa construção:

* grafo completo: quando existe uma, e somente uma aresta para cada par distinto de vértices;
* grafo trivial: é um grafo com unicamente um vértice.

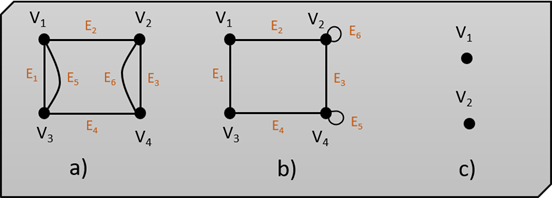
Assim como tínhamos o grau de cada nó de uma árvore, em grafos também podemos encontrar o **grau de cada nó (vértice)**. O grau de um vértice nada mais é do que a soma de todas as arestas incidentes nele. Por exemplo, na Figura 3 (a) anterior, todos os vértices têm grau 2, já na Figura 3 (c) temos vértices de grau 2 e outros chegando a grau 5, como o vértice 3.

Dentro de algumas particularidades de construção de grafos, podemos citar as **arestas múltiplas**, que são aquelas que estão conectadas nos mesmos vértices. Na Figura 4 (a), a seguir, temos essa situação. Os vértices 1 e 3 estão conectados por duas arestas (aresta 1 e 5).

Na Figura 4 (b), temos um vértice com **laços**.Um laço acontece quando uma aresta contém somente um vértice para se conectar, iniciando e terminando nele. Vemos isso na aresta 5 e 6 da Figura 4 (b).

Por fim, vemos um **grafo desconexo** (Figura 4, c). Nesse tipo de grafo, temos pelo menos um vértice sem nenhuma aresta. No exemplo da Figura 4, ambos estão sem arestas, mas isso não é obrigatório.

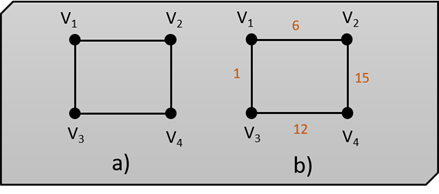
Figura 4 – Peculiaridades na construção de grafos

Crédito: Vinicius Pozzobon Borin.

Finalizando nossos conceitos teóricos, precisamos entender que podemos atribuir pesos para nossas arestas. O peso da aresta representa um custo atrelado àquele caminho. Por exemplo, vamos voltar ao nosso exemplo de mapeamento de estradas. Caso desejarmos traçar uma rota de um ponto A até outro ponto B da cidade mapeada, os pesos nas arestas poderiam ser calculados por meio do tráfego de veículos da rua, por exemplo. Enquanto mais veículos, maior o peso da aresta, e pior o desempenho dela na escolha da rota.

Quando não indicamos nenhum peso nas arestas, assumimos que todas elas têm o mesmo valor (Figura 5, a). Quando damos um número para cada aresta, indicamos os valores no próprio desenho do grafo (Figura 5, b). Chamamos um grafo com pesos em arestas de **grafo ponderado**.

Figura 5 – Peculiaridades na construção de grafos

Crédito: Vinicius Pozzobon Borin.

**1.2 APLICAÇÕES DE GRAFOS**

Acerca da aplicabilidade da estrutura de dados do tipo grafo*,*a gama de aplicações é bastante grande. Citamos:

* encontrar rotas e melhores trajetos em mapas;
* escrever algoritmos de inteligência artificial que calculam o menor número de movimentos necessários para a vitória em uma partida de damas, ou xadrez;
* mapear um jogo de tabuleiro para criar jogadas e movimentos;
* encontrar algo bastante próximo de nós, como o médico mais próximo conveniado ao nosso plano de saúde;
* conexões e tabelas de roteamento em redes de computadores;
* mapeamento de interações em redes sociais.

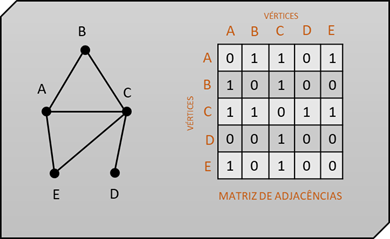
**TEMA 2 – REPRESENTAÇÃO DE GRAFOS**

Vamos apresentar duas maneiras distintas e bastante comuns de representação. Todas aqui apresentadas poderão ser implementadas em programação.

**2.1 MATRIZ DE ADJACÊNCIAS**

A representação de um grafo por uma matriz de adjacências consiste em criar uma matriz quadrada de dimensão V, em que V será o número de vértices do grafo. Por exemplo, se o grafo contém 10 vértices, teremos uma matriz 10x10, não importando o número de arestas. Na Figura 6 a seguir, há a ilustração de um exemplo de matriz de adjacências.

Figura 6 – Exemplo de matriz de adjacências

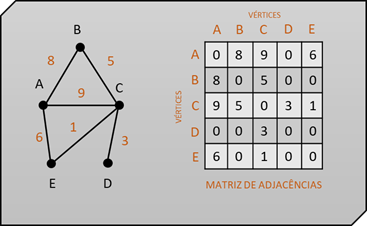
Crédito: Vinicius Pozzobon Borin.

Assim como na representação anterior, povoamos nossa matriz com valores 0 ou 1. A análise que fazemos para preenchimento da matriz é efetuada de uma forma diferente agora; observamos cada uma das linhas dos vértices e preenchemos na matriz:



Grafos podem, eventualmente, apresentar pesos em suas arestas, os quais são chamados de **grafos ponderados**. O emprego de uma matriz de adjacências facilita o uso para grafos ponderados, pois o peso de cada aresta é explicitado na estrutura. Vejamos a Figura 7 a seguir.

Figura 7 – Exemplo de matriz de adjacências para grafo ponderado

Crédito: Vinicius Pozzobon Borin.

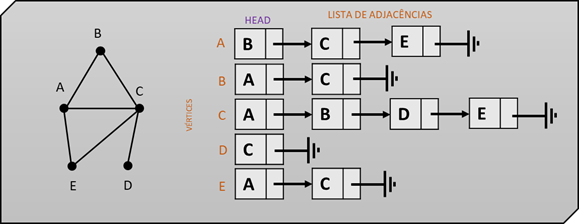
**2.2 LISTA DE ADJACÊNCIAS**

A representação de um grafo por uma lista de adjacências é muito adotada no meio de programação, pois trabalha com listas encadeadas e manipulação de ponteiros de dados.

A Figura 8 a seguir contém um exemplo de lista de adjacências. A proposta dessa representação é a de criar um *array* do tamanho da quantidade de vértices existentes no grafo. Em cada posição desse *array* teremos uma lista encadeada contendo os endereços dos “vizinhos” daquele vértice. Conceitualmente, **vizinhos de um vértice são todos os outros vértices que estão conectados a ele**.

Assim, teremos uma lista encadeada de vizinhos para cada posição do *array* de vértices criado. Na lista, cada vizinho apontará para outro vizinho daquele mesmo nó.

Figura 8 – Exemplo de lista de adjacências

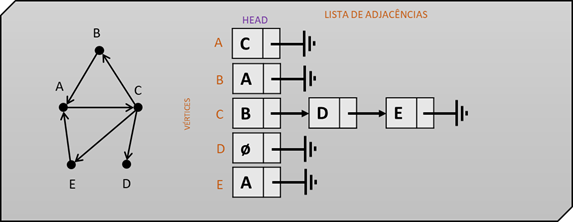
Crédito: Vinicius Pozzobon Borin.

No exemplo da Figura 8 anterior, teremos cinco vértices, portanto, podemos ter um *array* de dimensão 5, em que cada posição do array terá o endereço do *head* (também vizinho) para uma lista encadeadas de vizinhos daquele vértice.

Observemos o vértice A: ele está na primeira posição do *array* (posição zero), assim como contém como vizinho o vértice B, vértice C e o vértice E. Assim, na posição zero do *array* temos o endereço para um dos vizinhos de A. Esse vizinho será o *head*de uma lista encadeada. O vizinho escolhido para ser o *head* foi o vértice B. Assim, B apontará para o outro vizinho, C, que por sua vez aponta para E. Teremos uma lista encadeada de vizinho do vértice A. De forma análoga, criamos uma lista encadeada de vizinhos para cada vértice existente no grafo.

Grafos também podem ser do tipo dirigido. Todos os exemplos que vimos até então são não dirigidos. Um **gafo dirigido**é aquele que contém um único sentido de conexão entre os vértices e representamos esse sentido por uma seta. Vejamos a Figura 9 a seguir.

Figura 9 – Exemplo de lista de adjacências em grafos dirigidos

Crédito: Vinicius Pozzobon Borin.

É interessante notar que representamos como vizinho de cada vértice somente aqueles os quais respeitam os sentidos das setas. O vértice D não tem vizinhos, pois sobre ele há somente índice uma aresta vinda de C.

Embora seja possível implementar um grado dirigido com matriz de adjacências, o mais usual é o emprego da lista de adjacências.

**2.3 COMPARATIVO DAS REPRESENTAÇÕES**

Podemos comparar o desempenho das representações de acordo com a quantidade de vértices e arestas contidas no grafo. Temos um **grafo denso**quando a quantidade de aresta E é equivalente ao quadrado da quantidade de vértices V, e um **grafo esparso** quando o total de arestas for igual ao de vértices:

* Grafo denso: 
* Grafo esparso: 

Em uma matriz de adjacência, sempre utilizaremos de espaço na memória. Ou seja, se tivermos um grafo com 10 mil vértices, teríamos 100.000.000 *bytes* de uso de memória. Portanto, uma desvantagem dessa representação é o excesso de uso de memória, especialmente para grafos grandes.

Em termos de complexidade de tempo de execução, a matriz é mais rápida que a lista de adjacência para grafos densos, pois acessa qualquer dado com o mesmo tempo. Por fim, essa representação é mais simples de ser aplicada para grafos ponderados.

Já uma lista de adjacência é mais rápida se comparada à matriz, e emprega menos espaço de memória para grafos esparsos. Em contrapartida, para grafos bastante densos são mais lentas do que a versão matricial, isso porque grafos densos tendem a formar grandes listas de vizinhos, que quando representadas por listas encadeadas geram um tempo de acesso ao dado bastante elevado.

**TEMA 3 – BUSCA EM PROFUNDIDADE**

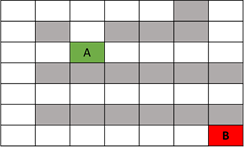
Após entendermos como construir um grafo, vamos analisar como caminhamos pelos elementos dessa estrutura de dados. Então, vamos assumir que temos um grafo já construído utilizando uma lista de adjacências.

Agora, queremos andar por cada vértice do grafo, sem repetir nenhum, partindo de um vértice de origem. Cada vértice visitado é marcado para que não seja revisitado. O algoritmo é encerrado quando o último vértice é visitado.

O algoritmo de busca em profundidade, mais conhecido pelo seu nome inglês ***Depth-First Search* (DFS)**, apresenta uma lógica bastante simples e funcional para resolver esse problema de descoberta do grafo.

Vamos acompanhar o funcionamento desse algoritmo com um exemplo. Para um melhor entendimento inicial, adotaremos uma analogia de um labirinto. Na Figura 10 a seguir, temos um labirinto representado por um tabuleiro 7x7. Queremos partir de um ponto A e chegar em um ponto B.

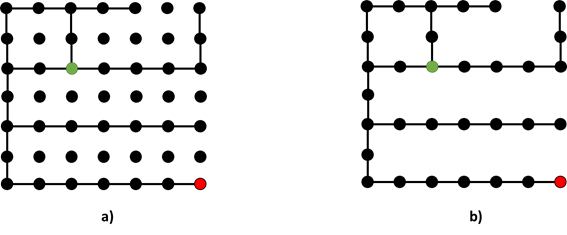
Figura 10 – Labirinto 7x7 para exemplo

Crédito: Vinicius Pozzobon Borin.

Podemos entender que cada quadradinho do labirinto pode ser considerado um vértice. Se houver conexão entre dois quadradinhos, ou seja, se eles se encostam, existe, portanto, uma aresta de conexão entre eles.

Não existe aresta nos quadradinhos marcados em cinza. Esses quadrados são considerados paredes em nosso labirinto. Podemos representar o labirinto como um grafo quadrado de 7x7 vértices, conforme Figura 11 (a) a seguir. Note que alguns vértices ficam sem conexão, pois são as paredes. Para fins de simplificação, podemos eliminar esses vértices do grafo (Figura 11, b).

Figura 11 – Grafo representativo do labirinto 7x7

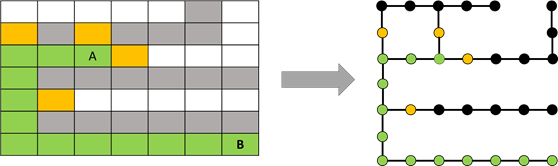
Crédito: Vinicius Pozzobon Borin.

Precisamos agora partir do ponto A, em verde, até chegar no ponto B, em vermelho utilizando o algoritmo de busca em profundidade. Lembre-se de que cada vértice só tem o conhecimento de quem são seus vizinhos. Portanto, ele não tem a menor ideia de que caminho deve seguir para chegar em B.

A DFS faz o seguinte procedimento: como cada vértice tem uma lista de vizinhos, a DFS simplesmente acessa o primeiro vizinho da lista daquele vértice e avança para ele. No novo vértice, acessa-se o primeiro vizinho novamente, e assim por diante, ou seja, vamos acessando o primeiro vizinho de cada lista até atingirmos o nosso objetivo.

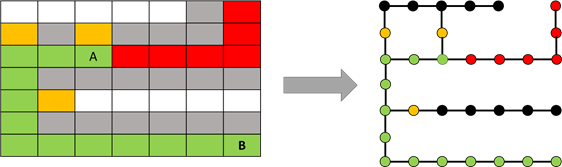
Esta lógica pode nos gerar situações com desempenhos bastantes distintos. Imaginemos que, por coincidência, os primeiros vizinhos de cada vértices sigam exatamente em direção ao ponto B. Faremos um caminho direto em direção ao ponto B. Vejamos a Figura 12 a seguir.

Figura 12 – Labirinto indicando o trajeto em verde de A até B: em amarelo, os outros vizinhos não acessados

Crédito: Vinicius Pozzobon Borin.

Todavia, podemos ter uma situação oposta. Ou seja, imaginemos que, partindo do ponto A, o primeiro vizinho seja para um caminho oposto. Então, percorreríamos o trajeto até um ponto sem saída, conforme pode ser visto na Figura 13 a seguir. Somente ao atingirmos o ponto final do trajeto é que retornamos para o último vértice com trajetos disponíveis e tomamos outro caminho, este em direção ao ponto B, por exemplo.

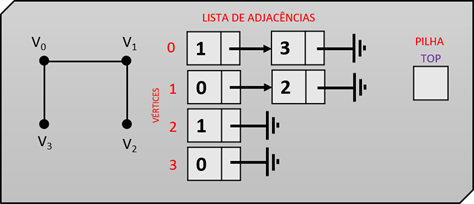
Figura 13 – Labirinto indicando o trajeto em verde de A até B

Crédito: Vinicius Pozzobon Borin.

Por meio do exemplo do labirinto foi possível notar que o algoritmo de busca em profundidade escolhe um caminho e vai profundamente percorrendo ele até tentar outro trajeto.

Vamos agora investigar um grafo mais simples com o objetivo de compreendermos o algoritmo da DFS. Na Figura 14 a seguir, vemos o grafo que vamos utilizar.

Figura 14 – Busca em profundidade no grafo: estado inicial

Crédito: Vinicius Pozzobon Borin.

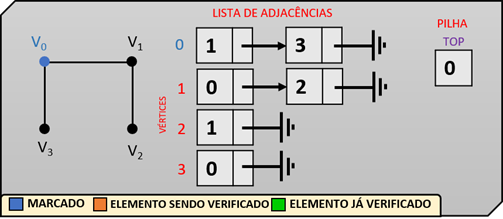
Queremos andar por todos os vértices, partindo de um vértice de origem. O primeiro vizinho detectado (mais na frente na lista encadeada daquele vértice) será imediatamente acessado.

Assim, cada novo elemento ainda não descoberto é selecionado com base nas listas encadeadas de cada vértice até que o último seja encontrado. Os elementos descobertos vão sendo empilhados e desempilhados seguindo as regras de uma estrutura do tipo pilha. Ou seja, só podemos voltar ao vizinho de um nó mais abaixo na pilha se resolvermos primeiro o elemento mais acima, no topo da pilha.

Vamos analisar o grafo da Figura 14 anterior. Mostraremos como se dá o funcionamento lógico desse algoritmo e mostraremos graficamente a descoberta do grafo. O grafo contém quatro vértices iniciando em zero e três arestas. Podemos construir a lista de adjacências desse grafo. Ainda, é importante observar que temos uma pilha que, neste momento, está vazia.

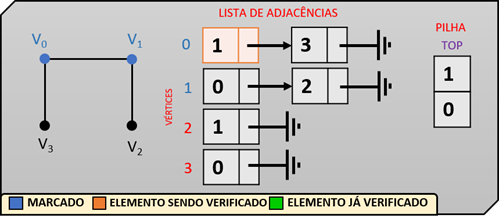
Desejamos iniciar a descoberta de nosso grafo pelo vértice zero. Na Figura 15 a seguir, vemos que esse vértice é então marcado como já visitado (cor azul). Assim, não é possível mais revisitá-lo. Em nossa pilha, que estava vazia, colocamos nosso vértice inicial zero.

Figura 15 – Busca em profundidade no grafo, partindo de: etapa 1

Crédito: Vinicius Pozzobon Borin.

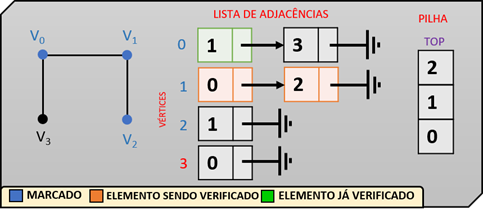
Como esse vérticeestá no topo da pilha, vamos acessar sua respectiva lista de vizinhos. Acessamos, então, o primeiro elemento da lista encadeada de vizinhos de , que é o vértice (destacado em laranja na Figura 16 a seguir). Imediatamente, colocamos esse vértice como já visitado (cor azul) no grafo e inserimos ele no topo da pilha, acima de . Agora, temos dois dos quatro vértices já visitados.

Figura 16 – Busca em profundidade no grafo, partindo de: etapa 2

Crédito: Vinicius Pozzobon Borin.

O próximo vértice a ser visitado será o primeiro vizinho da lista encadeada do vértice . É importante perceber que nem terminamos de varrer toda a lista encadeada de , mas já pulamos para a lista de outro vértice. O primeiro vizinho de é o próprio , que na verdade já foi visitado. Nesse caso, passamos para o próximo elemento da lista encadeada, que é o vértice . Este, ainda não visitado (em laranja), é então marcado como visitado e colocado no topo da pilha, que agora contém três elementos. Vejamos a Figura 17 a seguir.

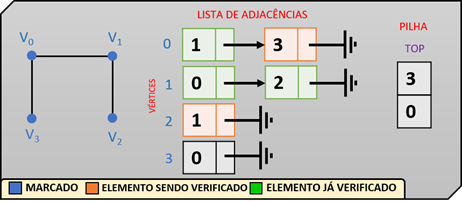
Figura 17 – Busca em profundidade no grafo, partindo de: etapa 3

Crédito: Vinicius Pozzobon Borin.

Nesse momento, estamos na lista encadeada do vértice . Na Figura 18 a seguir, vemos que precisamos acessar o primeiro vizinho do vértice (em laranja); esse vizinho é o vértice , que já foi visitado. Ainda, toda a lista de vizinhos do vértice já foi verificada, não havendo mais nenhum vizinho de .

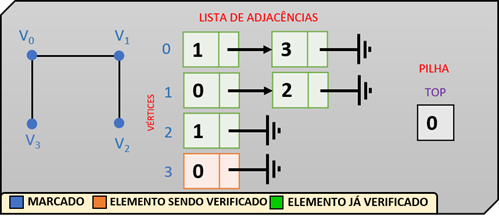
Desempilhamos o vértice atual () e voltamos para o elemento abaixo dele na pilha (), que já teve todos os seus vizinhos visitados. Desempilhamos ele, voltamos para e acessamos o segundo elemento da lista encadeada, pois o primeiro havia sido visitado anteriormente. Encontramos, assim, o último vértice não visitado, o . Marcamos ele e colocamos no topo da pilha.

Figura 18 – Busca em profundidade no grafo, partindo de: etapa 4

Crédito: Vinicius Pozzobon Borin.

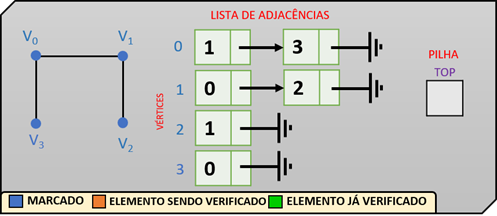
Todos os vértices já haviam sidos visitados na etapa anterior. Porém, ainda precisamos encerrar nosso algoritmo. Na Figura 19 a seguir, no vértice , verificamos o primeiro vizinho dele na lista encadeada, o vértice , anteriormente marcado. Chegamos ao final da lista; desempilhamos , restando somente .

Figura 19 – Busca em profundidade no grafo, partindo de: etapa 5

Crédito: Vinicius Pozzobon Borin.

Embora já tenha sido visitado, faltava desempilhá-lo (Figura 20). Fazendo isso, temos nossa pilha vazia e todos os quatro vértices visitados, encerrando nosso algoritmo de busca por profundidade.

Figura 20 – Busca em profundidade no grafo, partindo de: etapa 6

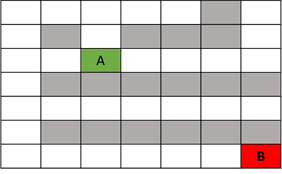
Crédito: Vinicius Pozzobon Borin.

**TEMA 4 – BUSCA EM LARGURA**

O algoritmo de busca em largura, mais conhecido pelo seu nome inglês ***Breath-First Search* (BFS)**, trabalha com a ideia de visitar primeiramente todos os vizinhos de um mesmo vértice atualmente marcado antes de pular para o próximo vértice.

Vamos acompanhar o funcionamento desse algoritmo com um exemplo. Para um melhor entendimento inicial, adotaremos uma analogia de um labirinto. Na Figura 21 a seguir, temos um labirinto representado por um tabuleiro 7x7. Queremos partir de um ponto A e chegar em um ponto B.

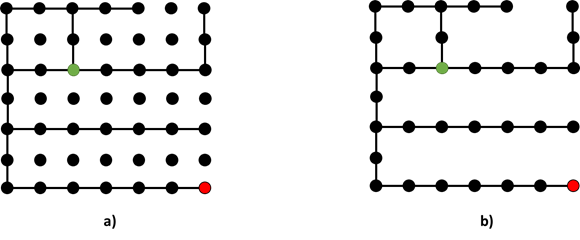
Figura 21 – Labirinto 7x7 para exemplo

Crédito: Vinicius Pozzobon Borin.

Podemos entender que cada quadradinho do labirinto pode ser considerado um vértice. Se houver conexão entre dois quadradinhos, ou seja, se eles se encostam, existe, portanto, uma aresta de conexão entre eles.

Não existe aresta nos quadradinhos marcados em cinza. Esses quadrados são considerados paredes em nosso labirinto. Podemos representar o labirinto como um grafo quadrado de 7x7 vértices, conforme Figura 22 (a) a seguir. Notemos que alguns vértices ficam sem conexão, pois são as paredes. Para fins de simplificação, podemos eliminar esses vértices do grafo (Figura 22, b).

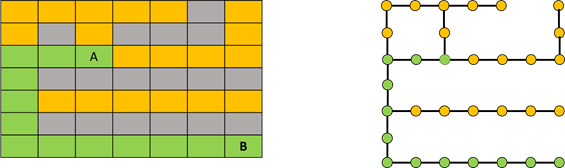
Figura 22 – Grafo representativo do labirinto 7x7

Crédito: Vinicius Pozzobon Borin.

Precisamos agora partir do ponto A, em verde, até chegar ao ponto B, em vermelho, utilizando um algoritmo de busca em largura. A BFS faz o seguinte procedimento: ao invés de tomar um caminho como único e seguido até o final, a BFS anda para todos os lados simultaneamente. Ela anda um pouco em cada um dos caminhos possíveis.

Vejamos a Figura 23, a seguir. Partindo do ponto A, temos três caminhos possíveis: esquerda, direta e cima. A BFS vai andar para todas essas direções simultaneamente. Dessa maneira, deixamos marcado em amarelo todos os caminhos traçados.

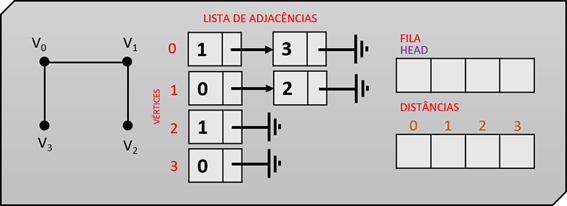
Figura 23 – Grafo representativo do labirinto 7x7

Crédito: Vinicius Pozzobon Borin.

Uma diferença interessante em relação à busca por profundidade é que ele trabalha com uma estrutura do tipo fila para indicar qual o próximo vértice a ser acessado, já na DFS tínhamos uma estrutura de pilha.

Vamos acompanhar o funcionamento desse algoritmo com um exemplo. Na Figura 24, a seguir, está ilustrado o grafo que será trabalhado. É interessante perceber que temos agora um *array* de distâncias, que manterá armazenada a distância de cada vértice em relação ao vértice de origem, ajudando na decisão de qual será o próximo vértice a ser visitado.

Figura 24 – Busca em largura no grafo, partindo de: estado inicial

Crédito: Vinicius Pozzobon Borin.

A proposta desse algoritmo é a de partir de um vértice de origem e acessar a lista de adjacências daquele vértice, ou seja, seus vizinhos. Partindo da lista de vizinhos, calcula-se a distância deles para o vértice de origem e salva-se em um vetor de distâncias. Os elementos vão sendo enfileirados à medida que vão sendo acessados nas listas encadeadas.

Iniciamos nossa análise no vértice novamente. Nesse modo, podemos imediatamente marcá-lo como já visitado (cor azul). Colocamos também esse vértice na fila. Lembre-se de que, em uma fila, a inserção acontece sempre no final dela (Fifo). Como a fila estava vazia, o inserimos na primeira posição. A distância do vértice de origem para ele mesmo será sempre zero (Figura 25).

Figura 25 – Busca em largura no grafo, partindo de: etapa 1

Crédito: Vinicius Pozzobon Borin.

Conhecendo a lista de vizinhos do vértice , devemos passar elemento por elemento dessa lista encadeada, inserindo-os ao final da fila e calculando as distâncias.

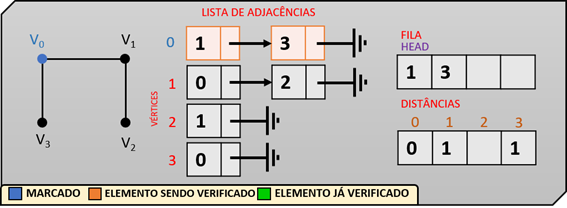
Na Figura 26, a seguir, encontramos o primeiro vizinho de , que é . Colocamos na fila e calculamos a distância dele para a origem. O cálculo da distância é feito usando o valor da distância do vértice atual (valor zero) e acrescentando uma unidade, resultando em distância um (), conforme Figura 26, a seguir.

Figura 26 – Busca em largura no grafo, partindo de: etapa 2

Crédito: Vinicius Pozzobon Borin.

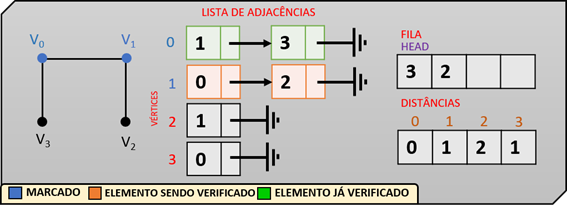
Na Figura 27, a seguir, passamos para o próximo vizinho do vértice . O vértice recebe o mesmo tratamento que : sua distância é de valor um , pois está a um salto de distância da origem, e ele é colocado no final da fila, após o vértice Assim, encerramos a varredura dos vizinhos do vértice e calculamos as distâncias, mas ainda não visitamos eles.

Figura 27 – Busca em largura no grafo, partindo de: etapa 3

Crédito: Vinicius Pozzobon Borin.

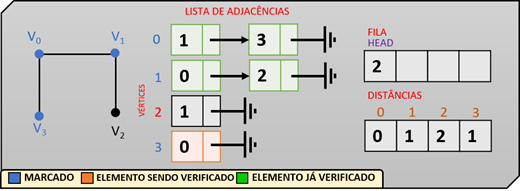
Na Figura 28, a seguir, com encerrado (cor verde), vamos para o próximo vértice da fila, o vértice . Marcamos ele, o removemos da fila e acessamos sua lista encadeada de vizinhos. O primeiro vizinho é o vértice , que já foi visitado, portanto, seguimos para o próximo vértice, que é Esse vértice ainda não foi visitado, então colocamos ele ao final da fila e calculamos uma distância. Como ele está a dois vértices de distância da origem, sua distância é dois ().

Figura 28 – Busca em largura no grafo, partindo de: etapa 4

Crédito: Vinicius Pozzobon Borin.

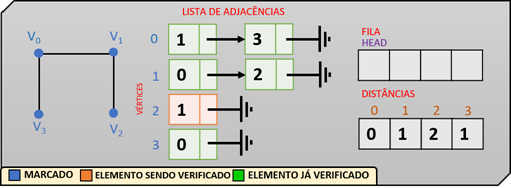
Na etapa 4, já encerramos os cálculos de todas as distâncias possíveis e enfileiramos todos os vértices restantes, e . Assim, na etapa 5 (Figura 29), removemos da fila o próximo vértice e marcamos ele (cor azul). Acessamos, então, a sua lista de vizinhos. Nela, só existe o vértice , já acessado. Portanto, nada mais fazemos nessa etapa e podemos passar para o próximo elemento da nossa fila.

Figura 29 – Busca em largura no grafo, partindo de: etapa 5

Crédito: Vinicius Pozzobon Borin.

Na Figura 30, a seguir, acessamos o vértice , último ainda não descoberto, e vemos que ele tem como vizinho somente o vértice . Como o vértice 1 já é conhecido, nada mais fazemos nessa etapa.

Figura 30 – Busca em largura no grafo, partindo de: etapa 6

Crédito: Vinicius Pozzobon Borin.

Por fim, na etapa 7 (Figura 31), estamos com todos nossos elementos já visitados e todas as listas encadeadas varridas. Nosso algoritmo se encerra aqui.

Figura 31 – Busca em largura no grafo, partindo de: etapa 7

Crédito: Vinicius Pozzobon Borin.

**4.1 COMPARATIVO DE DESEMPENHO: DFS X BFS**

A única e real diferença no funcionamento de ambas as buscas no grafo é que a BFS opera com uma estrutura de fila, já a DFS com uma pilha. Então qual o impacto disso na complexidade *Big-O*?

A análise não é trivial, mas vamos assumir que |V| é nosso total de vértices, e que |E| é o total de arestas. Inicialmente, independentemente do algoritmo escolhido, precisamos inicializar todos nossos vértices. Se considerarmos esse tempo de inicialização, temos para tal um O(|V|).

No BFS, com a fila, fazemos inserções e remoções sem dependência do tamanho do conjunto de dados, ou seja, o acesso ao dado, como inserção e remoção, é sempre O(1). Todavia, nesse algoritmo precisamos verificar as distâncias de um vértice com todos os seus vizinhos, portanto, em um pior cenário, temos esse procedimento sendo realizado para todas as arestas: O(|E|). Como fazemos esse cálculo uma só vez para todo o algoritmo, ficamos com O(|E|). Juntando a inicialização com a complexidade do BFS, temos: O(|V| + |E|).

De maneira bastante similar, o DFS opera com uma pilha e também tem tempo constante O(1), mas realizado o procedimento para cada uma das arestas. Sendo assim, a complexidade da DFS também será O(|V| + |E|).

**TEMA 5 – CAMINHO MÍNIMO (ALGORITMO DE DIJKSTRA)**

Vamos agora estudar um algoritmo que, por meio de um grafo conhecido, encontra a menor rota (caminho com menor peso) entre dois vértices.

O algoritmo investigado neste tópico será o algoritmo de Dijkstra, que tem esse nome em razão do cientista da computação holandês Edsger Dijkstra, que publicou o algoritmo pela primeira vez em 1959. Esse algoritmo é o mais tradicional para realizar a tarefa de encontrar um caminho. Para realizar tal procedimento, podemos utilizar um **grafo ponderado**, ou seja, aquele com pesos distintos nas arestas.

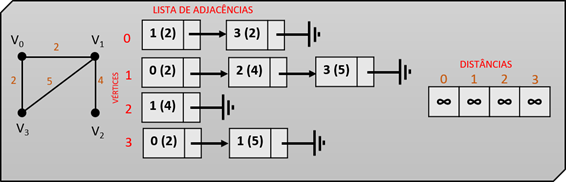
Vamos aplicar o algoritmo de Dijkstra em um grafo ponderado e obter as menores rotas, partindo de uma origem para todos os outros vértices desse grafo. Todo o algoritmo será apresentado utilizando uma **métrica aditiva**. Isso significa que essa métrica vai encontrar a menor rota considerando o **menor peso somado entre os caminhos**.

Um exemplo prático de métrica aditiva seria o tráfego de veículos em rodovias. Enquanto maior o tráfego, pior o desempenho daquela aresta, e, portanto, maior o seu peso no grafo.

Para conhecimento, existem outros tipos de métricas, como a multiplicativa, em que o melhor caminho encontrado é aquele cujo produto entre os caminhos é o maior valor, e os pesos são multiplicativos nessa situação. Um exemplo dessa métrica poderia ser o limite de velocidade das estradas: enquanto maior o limite, mais rápido o veículo anda e, portanto, melhor aquela rota/aresta.

Na Figura 32, a seguir, temos o estado inicial da lista de adjacências ponderadas. Os valores que estão entre parênteses são os pesos das arestas. Por exemplo, na lista encadeada do vértice , temos como primeiro elemento e o peso da aresta entre eles está indicado pelo valor 2 entre parênteses.

Figura 32 – Algoritmo de *Dijkstra*, partindo de: estado inicial

Crédito: Vinicius Pozzobon Borin.

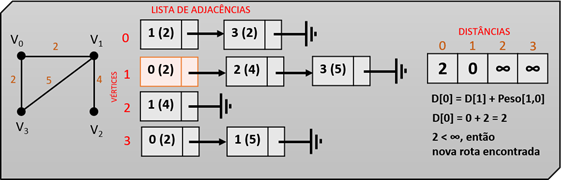
O algoritmo do caminho mínimo precisa calcular as menores rotas de um vértice em relação a todos os outros. Portanto, um vetor com distâncias fica armazenado. O estado inicial desse vetor é que todas as rotas iniciam com um valor infinito, ou seja, como se o caminho de um vértice até o outro não fosse conhecido. À medida que os trajetos vão sendo calculados, os pesos das rotas vão sendo alterados.

Vamos explicar o funcionamento do algoritmo iniciando no vértice . Assim, encontraremos a rota de para todos os outros vértices existentes no grafo ponderado.

Na Figura 33, a seguir, iniciamos a primeira etapa de operação do método. Como partiremos de , já iniciaremos acessando a lista de vizinhos desse vértice e calculando as rotas para eles. Iniciamos encontrando a rota de para ele mesmo. Nesse caso, o vértice de origem para ele mesmo terá sempre peso zero, conforme apresentado no vetor de distâncias.

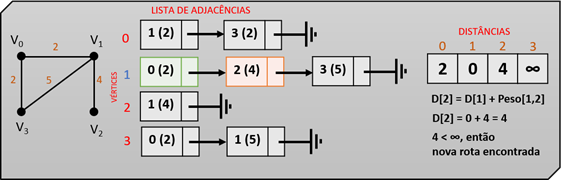
Em seguida, acessamos o *head*da lista encadeada de , que é , com um peso de aresta de 2. Assim, o cálculo da distância entre eles será o peso em acrescido do peso dessa aresta, resultando no valor 2. Esse valor, por ser menor do que infinito, é colocado no vetor na posição de . O cálculo está apresentado no canto inferior direito da Figura 33.

Figura 33 – Algoritmo de *Dijkstra*, partindo de: etapa 1

Crédito: Vinicius Pozzobon Borin.

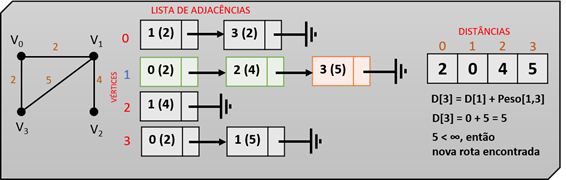
Seguimos na Figura 34, a seguir, acessando o segundo elemento da lista de vizinhos de . Em , fazemos o peso de acrescido da aresta para , resultando no valor 4, que por ser menor do que infinito é colocado como rota válida entre e .

Figura 34 – Algoritmo de *Dijkstra*, partindo de

Crédito: o autor.

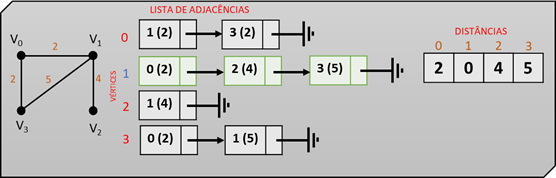
A etapa 3 está na Figura 35, a seguir. De forma semelhante às duas etapas anteriores, calculamos a distância entre e , resultando em 5, e colocamos no vetor de distâncias na posição de .

Figura 35 – Algoritmo de *Dijkstra*, partindo de: etapa 3

Crédito: Vinicius Pozzobon Borin.

Todos os vizinhos do vértice têm suas rotas calculadas. Um panorama geral é apresentado na Figura 36, a seguir, indicando que todos os vizinhos foram calculados (cor verde).

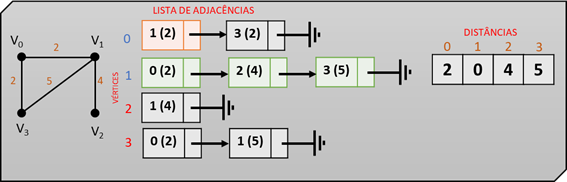
Figura 36 – Algoritmo de *Dijkstra*, partindo de: etapa 4

Crédito: Vinicius Pozzobon Borin.

Precisamos passar para o próximo vértice e calcular as rotas partindo de e passando por este vértice intermediário. O vértice que vamos assumir agora é o de menor caminho no vetor distâncias e que ainda não tinha sido marcado. Será, portanto, o vértice , com distância para sendo 2.

Seu primeiro vizinho é o próprio vértice de origem , na Figura 37, a seguir. Como esse vértice já foi visitado, podemos pulá-lo nessa análise.

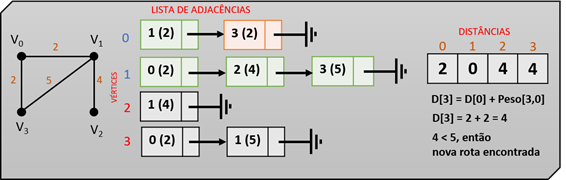
Figura 37 – Algoritmo de *Dijkstra*, partindo de: etapa 5

Crédito: Vinicius Pozzobon Borin.

Na Figura 38 adiante, seguimos para o próximo vizinho de , o . Isso significa que calcularemos uma rota . O peso dessa rota será o peso até , já calculado e de valor 2, somado com o peso da aresta e , que é 2 também. Assim .

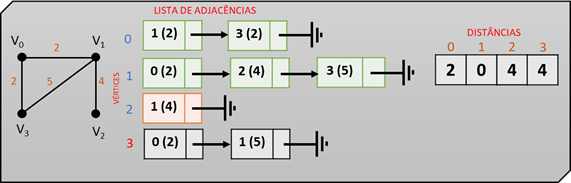
Observemos agora algo interessante. O peso da rota de até , passando por , resultou em peso 4. Anteriormente, tínhamos calculado o peso da rota direta entre e , cujo valor resultou em 5. Portanto, a rota passando por tem um peso menor (), resultando em uma nova rota até . Notemos que uma rota com peso menor, mesmo passando por um vértice a mais, acaba resultando em um caminho menor que o outro.

Figura 38 – Algoritmo de *Dijkstra*, partindo de: etapa 6

Crédito: Vinicius Pozzobon Borin.

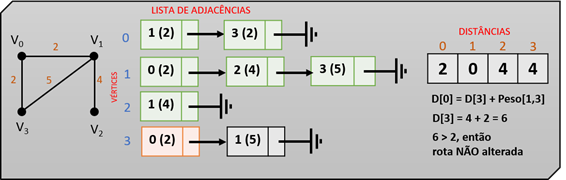
Na Figura 39 adiante, já encerramos nossos cálculos passando pelo vértice . Seguimos então para o próximo vértice não visitado e de menor distância no vetor, o vértice . O único vizinho de é vértice origem , já visitado. Portanto, não existirá uma nova rota aqui.

Figura 39 – Algoritmo de *Dijkstra*, partindo de: etapa 7

Crédito: Vinicius Pozzobon Borin.

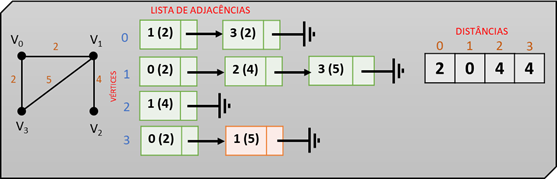
Na Figura 40 adiante, seguimos para o próximo, e último, vértice não visitado e de menor distância no vetor, o vértice . O primeiro vizinho de é , assim faremos o trajeto . O custo até é 4 e o peso de até é 2, resultando em 6, custo superior a atualmente a rota de 4.

Figura 40 – Algoritmo de *Dijkstra*, partindo de: etapa 8

Crédito: Vinicius Pozzobon Borin.

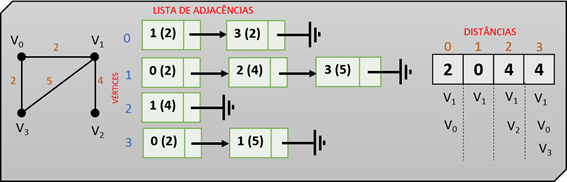
Na Figura 41, a seguir, temos o segundo vizinho de , o , que é a própria origem. Portanto, não precisamos analisar.

Figura 41 – Algoritmo de *Dijkstra*, partindo de: etapa 9

Crédito: Vinicius Pozzobon Borin.

Na Figura 42, a seguir, já percorremos todos os vizinhos de todos os vértices e calculamos todas as rotas possíveis. Desse modo, as distâncias resultantes estão colocadas no vetor de distâncias, e abaixo de cada valor está a sequência de vértices para aquela rota. Por exemplo, para atingirmos o vértice , a melhor rota encontrada foi , e não diretamente.

Figura 42 – Algoritmo de *Dijkstra*, partindo de: etapa 10

Crédito: Vinicius Pozzobon Borin.

**FINALIZANDO**

Nesta etapa, aprendemos sobre estrutura de dados do tipo grafo. Aprendemos que grafos não apresentam uma estrutura fixa na sua topologia de construção e são constituídos de vértices e arestas que conectam esses vértices.

Cada vértice do grafo conterá um conjunto de vértices vizinhos, os quais são vértices conectados por meio de uma única aresta. Aprendemos que podemos representar um grafo de duas maneiras distintas: matriz de adjacências e lista de adjacências. Esta última constrói um grafo utilizando diversas estruturas de listas encadeadas, em que cada vértice terá uma lista contendo todos os seus vizinhos.

Vimos também dois algoritmos distintos de descoberta do grafo, ou seja, como passear pelos vértices uma única vez, sem repeti-los. Para isso, vimos o algoritmo de busca por largura (BFS) e o de busca por profundidade (DFS).

Por fim, vimos um algoritmo clássico de cálculo de rotas dentro de um grafo. Ele calcula o menor trajeto dentro de um grafo pondero. O algoritmo estudado foi o de *Dijkstra*.

**REFERÊNCIAS**

ASCENCIO, A. F. G.; ARAÚJO, G. S. de. **Estruturas de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Halt 3, 2010.

BHARGAVA, A. Y. **Entendendo algoritmos**. São Paulo: Novatec, 2017.

CORMEN, T. H. **Algoritmos**: teoria e prática. 3. ed.São Paulo: Elsevier, 2012.

DROZDEK, A. **Estrutura de dados e algoritmos em C++**.Tradução da 4ª edição norte-americana. São Paulo: Cengage Learning Brasil, 2018.

FERRARI, R. et al. **Estruturas de dados com jogos**.São Paulo: Elsevier, 2014.

KOFFMAN, E. B.; WOLFGANG, P. A. T. **Objetos, abstração, estrutura de dados e projeto usando C++**. Porto Alegre: Grupo GEN, 2008.