**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.

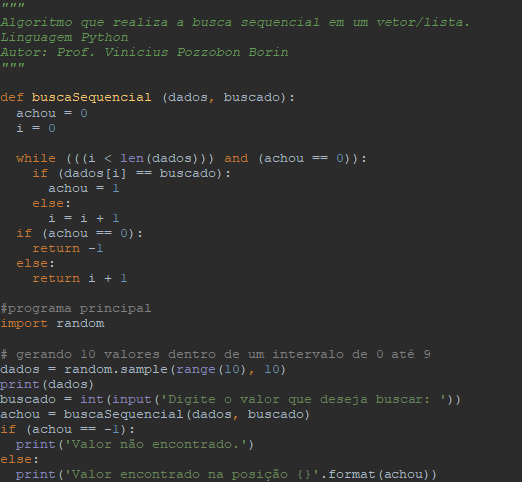
Figura 1 – Pesquisa sequencial: sequência de tentativas – em laranja, as tentativas; em verde, o acerto



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**.

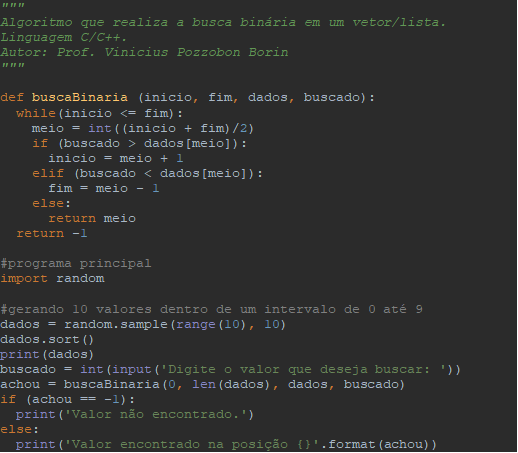
A Tabela 1 mostra todas as sete tentativas necessárias para localizar o dado 99 dentro do conjunto.

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.

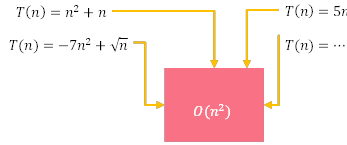
**2.1 NOTAÇÃO *BIG-O***

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*).

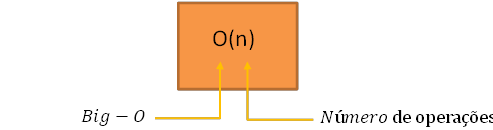
**Desse modo, para sabermos a complexidade de um algoritmo utilizando a notação *Big-O*, basta encontrarmos o termo de maior grau da equação que o descreve.**

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.

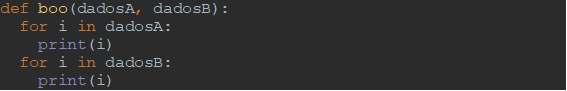
É importante compreender o que acontece quando um *loop* é executado. A cada iteração de um laço, todas as instruções dentro de sua estrutura serão executadas *n*vezes. Neste exemplo, *n,*é o tamanho de nosso vetor. Se o vetor/lista for de dimensão 10, o laço ocorre 10 vezes.



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:

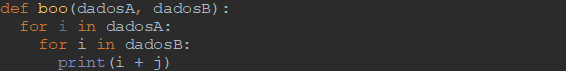


No *Big-O*, só nos interessa o termo de maior grau da equação. Portanto, negligenciamos o multiplicador dois na complexidade assintótica, ficando:



**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.

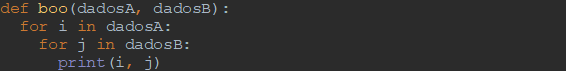


Em ações aninhadas, fazemos multiplicação:



**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.

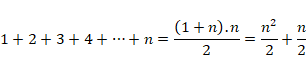
O caso mais comum é o de uma **série aritmética**, caracterizada como **uma sequência de números na qual a diferença entre dois termos consecutivos é constante**. Exemplos:

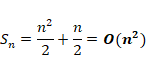

A soma dos *n*termos da progressão aritmética é dado por:



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 .



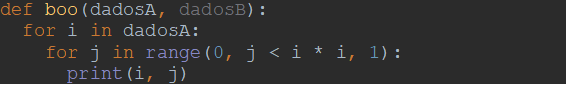
Um algoritmo com complexidade O(n²) normalmente terá dois laços de repetição aninhados. O laço da linha 4 depende do laço da linha 3. Se o tamanho do vetor for igual a 10, o segundo laço irá acontecer sempre 10 vezes o tamanho do vetor. Analisando o laço interno, temos uma PA constante, na qual:



**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.

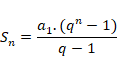
Tabela 2 – Variáveis dos laços que representam as iterações

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

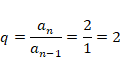
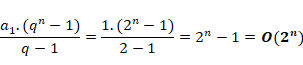
Vejamos outros exemplos de séries geométricas:


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:

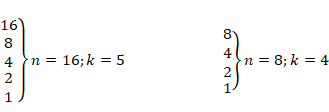
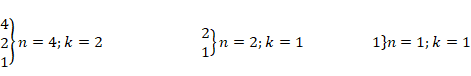
1. Descobrir o caso-base, que será sempre a menor parte possível para o problema;
2. Descobrir como reduzir o problema para que ele se torne o caso-base.

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**

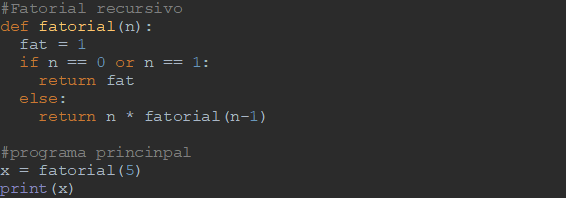
A complexidade de algoritmos recursivos demanda uma análise bastante minuciosa. De modo geral, para encontrarmos a complexidade de um algoritmo recursivo, devemos:

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**

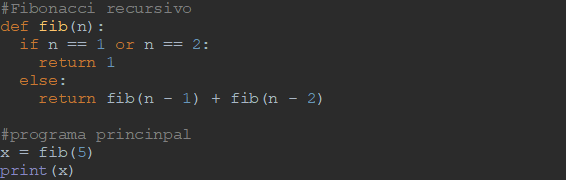
Existe uma equação que pode simplificar essa tarefa quando se trata de recursão. Porém, atente-se ao fato de que ela só é válida quando a quantidade de vezes que uma função recursiva invocar ela mesma duas ou mais vezes. A equação consiste em:



Restrições:

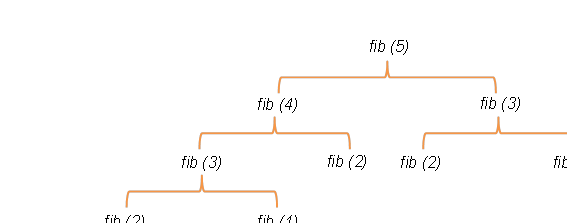
* O número de chamadas deve ser estritamente maior do que 1;
* Se o número de chamadas for igual a 1, teremos uma cadeia linear de chamadas (como na fatorial);
* Se a recursão ocorre em um *loop*, o número de chamadas depende de cada caso específico. Estudaremos isso melhor em breve.

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)

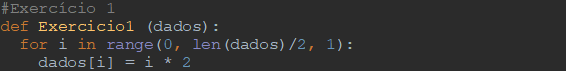
Portanto:



**5.2 EXERCÍCIOS**

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

**Exercício 1:**

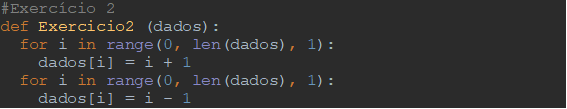


**Solução:**

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:**

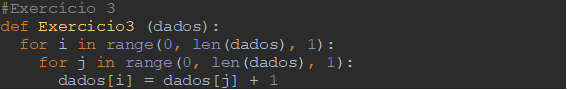


**Solução:**

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:**

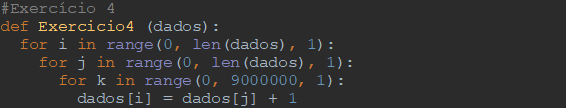
Temos um laço de repetição e, aninhado a ele, outro. Assim, existe dependência entre os laços. Nesse caso, precisamos verificar se as variáveis envolvidas no laço funcionam como uma PA, ou como uma PG.

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:



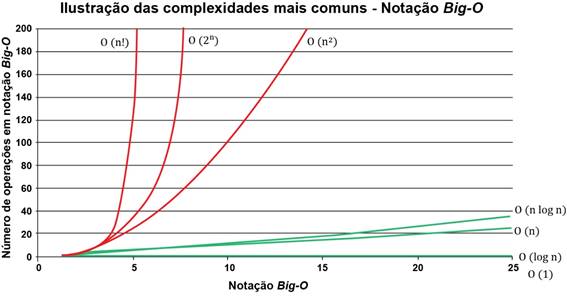
**FINALIZANDO**

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.**

**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).**

**Em seguida, realizamos um processo chamado de particionamento. Onde encontram-se os elementos menores do que o pivô e também os elementos maiores.**

**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.

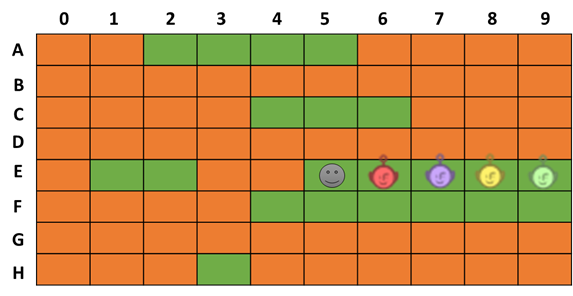
**TEMA 1 – ARRAYS E LISTAS ENCADEADAS**

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.

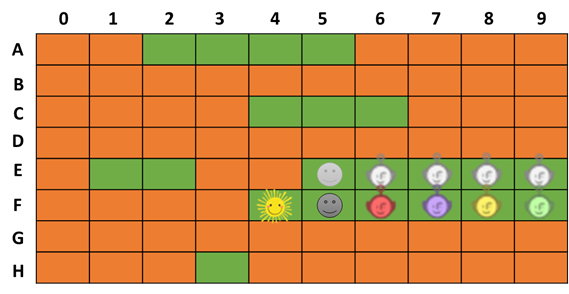
Essa situação em que temos mais um dado para ser colocado no array, mas sua inserção é negada, corresponde a um array do tipo estático, que nada mais é do que um conjunto de espaços sequenciais na memória, previamente reservado, mas que em hipótese alguma pode ter o seu tamanho alterado.

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



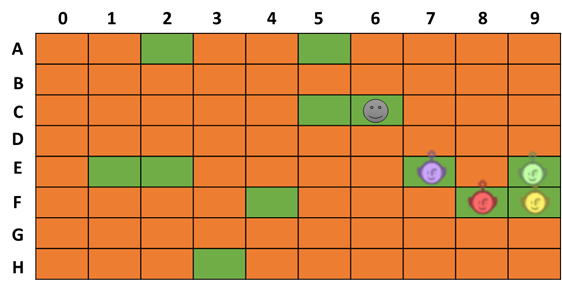
A analogia do cinema com assentos sem reserva, neste caso, corresponde a um array dinâmico, o qual também armazena dados sequencialmente. Agora, porém, à medida que surgem novos dados, é possível realocar o conjunto em outro espaço de memória em que caibam todos os dados.

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



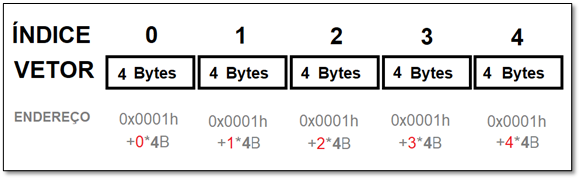
Solzinho, sempre atrasado, chega em cima do horário. Neste cenário, ele poderia facilmente sentar-se em qualquer outro local livre e assistir ao filme com seus amigos, na mesma sala de cinema. Esse tipo de configuração, de arranjo de dados sem necessidade de estarem sequencialmente alocados é chamado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



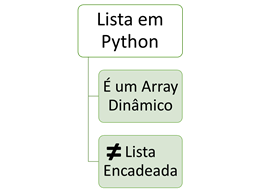
Podemos generalizar a forma de cálculo de cada posição na memória de um vetor pela equação 1:

|  |  |
| --- | --- |
|  | (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.

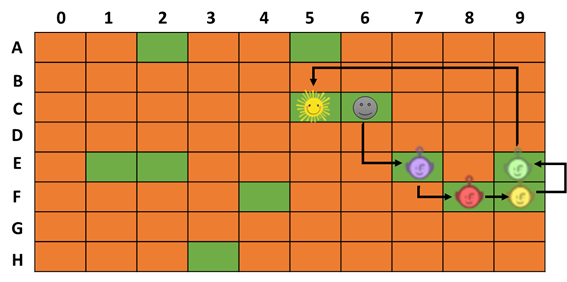
O **array dinâmico**é mais interessante, pois aloca somente o espaço sendo usado. Apesar disso, caso não exista na memória do programa nenhum bloco sequencial do tamanho desejado, haverá um problema, pois a alocação não acontecerá e o programa resultará em um erro de memória ou alocará o dado em local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

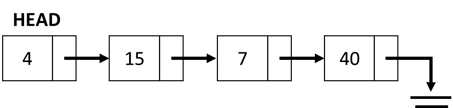
Um array, para ler um dado, independe do conjunto de dados, sendo O(1); a lista precisará varrer elemento por elemento até chegar no desejado, O(n).

**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

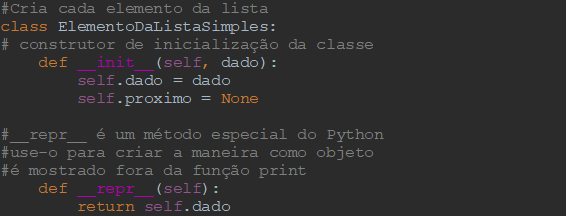


Note que o *head*contém o número 4 e aponta, ou seja, conhece o endereço do próximo elemento, o 15. Isso ocorre assim por diante, até chegarmos ao final desta lista, representada por um ponteiro nulo. Em suma as características das listas encadeadas são:

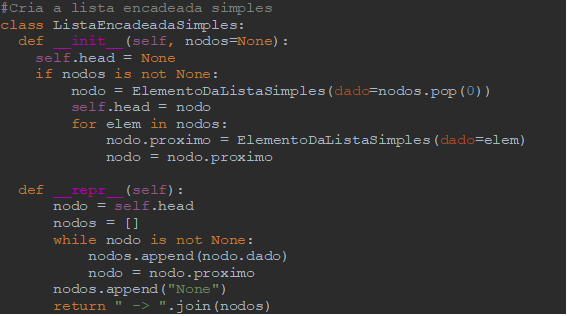
* sucessivos elementos conectados por ponteiros;
* o último elemento aponta para um endereço nulo, que é o final da lista;
* funciona dinamicamente, aumentando e diminuindo de tamanho conforme necessita;
* pode utilizar toda a memória destinada ao programa, uma vez que cada dado é capaz de ficar isolado 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:

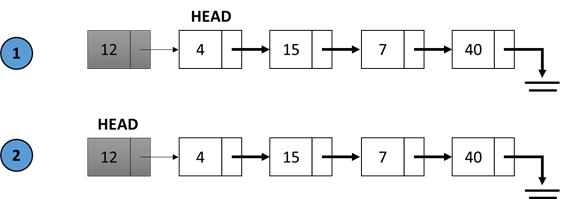


**2.2 INSERINDO NO INÍCIO DA LISTA ENCADEADA**

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



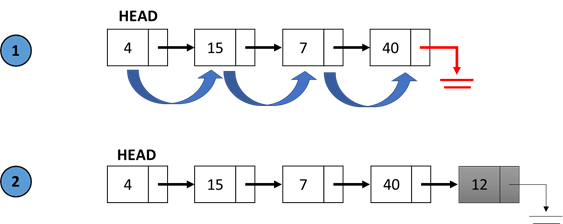
O código de inserção no início da lista é apresentado a seguir.



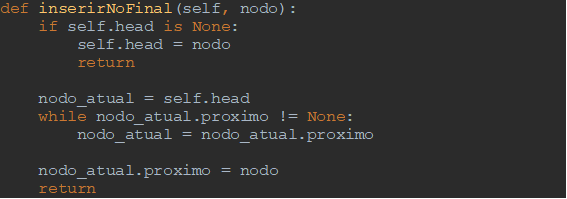
**2.3 INSERINDO NO FINAL DA LISTA ENCADEADA**

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**

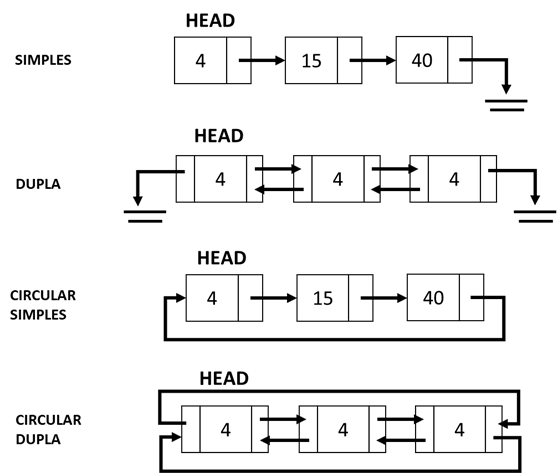
Existem diferentes tipos de listas encadeadas. Vamos conhecer um pouco mais sobre cada uma delas na teoria, embora estejamos focando nossos estudos somente na lista simples.

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).

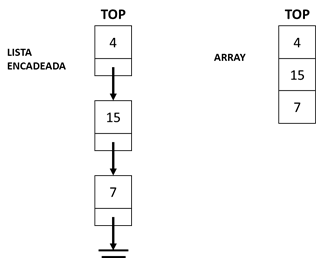
**3.1 CONSTRUINDO E MANIPULANDO UMA PILHA**

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.

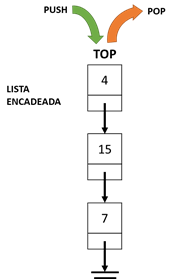
Se implementar a pilha como uma lista encadeada, o *top* seria o equivalente ao *head*, em que só poderíamos inserir ou remover dados diretamente do *head*. Ou seja, implementar uma pilha com listas encadeadas significa fazer uma inserção ou remoção somente no início dela.

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*.

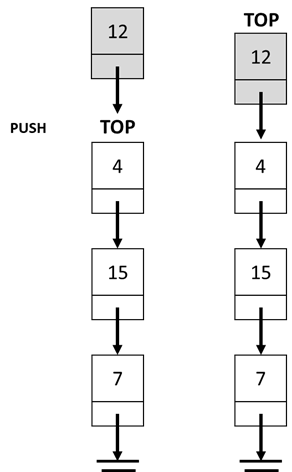
Figura 12 – *Push* e *pop* em pilhas com listas encadeadas



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.

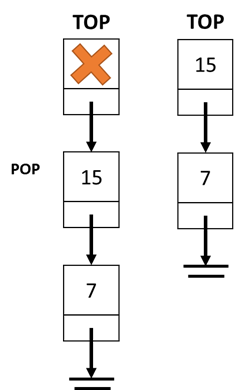
Percebeu que a sequência de passos para inserir na pilha é a mesmo para inserir no início de uma lista encadeada? De fato é o mesmo, só alteramos a terminologia de *head* para *top*. Na Figura 13 temos as duas etapas de inserção.

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.

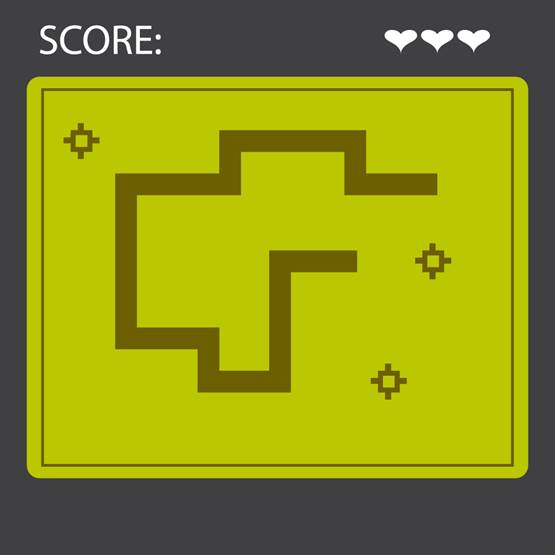
Ao abrir as portas do mercado, quem será o primeiro a entrar? Você! Note que, em uma fila, sempre quem chega primeiro é atendido primeiro, e quem chega por último, entra ao final da fila e será atendido por último. Uma estrutura de dados operando como uma fila, opera com o princípio de *o primeiro que 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*

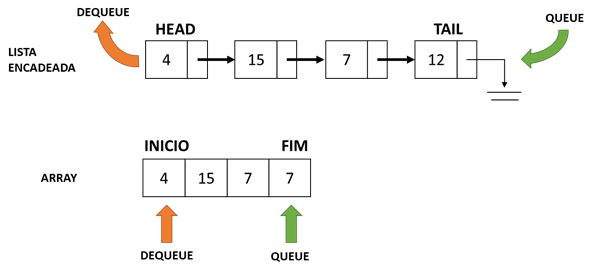
Crédito: 2DAssets / Shutterstock.

**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

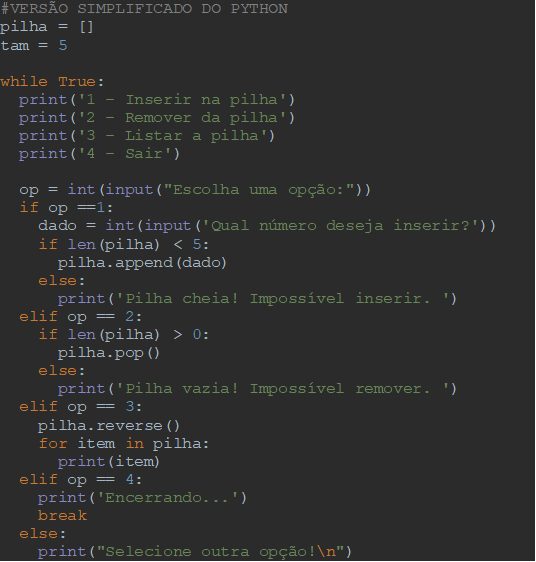


**4.2 IMPLEMENTANDO PILHAS E FILAS**

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.



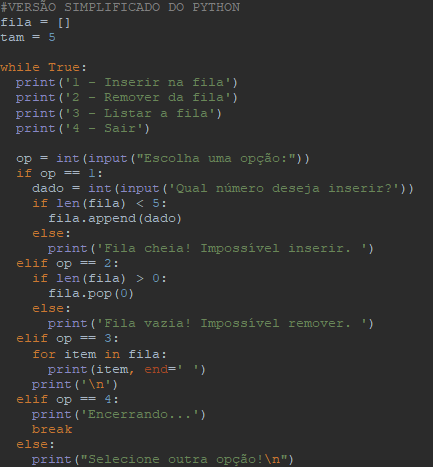
Note que criamos um código que contém um menu. Nele, o usuário é capaz de inserir um dado numérico na pilha, remover, mostrar a pilha na tela ou encerrar o programa. Como estamos trabalhando com uma lista em Python, podemos utilizar um método existente na linguagem chamado de *append*, que sempre adiciona (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)*.