ESTRUTURA DE DADOS

Matheus da Silva Serpa

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Métodos de pesquisa em listas: sequencial, binária e tabelas *hash*

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Diferenciar o funcionamento dos métodos de pesquisa sequencial e binário.
- Reconhecer o método de pesquisa por cálculo de endereço (tabela *hash*).
- Analisar algoritmos utilizando os métodos de pesquisa binário e sequencial.

Introdução

A quantidade de informação gerada e armazenada digitalmente cresce todo ano. Em 2013, o universo digital armazenava 4.4 zettabytes, e a expectativa para 2020 é um total de 44 zettabytes, e para 2025, um total de 463 exabytes gerados por dia (DESJARDINS, 2019).

Sistemas de informação, como sistemas bancários, redes sociais e mecanismos de busca, de forma geral, precisam lidar com uma grande quantidade de informações. Métodos de pesquisa são algoritmos que nos permitem recuperar informações em conjuntos de dados como esses.

Neste capítulo, você vai estudar sobre os métodos de pesquisa mais utilizados, que são a busca sequencial, a busca binária e a utilização de tabelas *hash* para armazenamento e pesquisa em dados. Você vai analisar as vantagens e desvantagens de cada método, tornando possível decidir qual é o melhor método para cada problema da vida real.

1 Métodos de busca sequencial e binária

A busca sequencial é o método de pesquisa mais simples e intuitivo. Para verificar se uma informação está contida em um conjunto de dados, percorremos este sequencialmente do início até que a informação seja encontrada, ou que o conjunto termine sem encontrar a informação.

Por exemplo, dado um vetor com N = 10 elementos, vamos buscar alguns valores. Primeiro, verificamos se o valor 55 está no conjunto de dados.

67	92	28	75	77	55	38	23	61	13

Vamos executar o algoritmo da esquerda para a direita (o inverso também é permitido e correto). Diversas comparações vão sendo feitas para verificar se o 55 é encontrado. A sequência de testes é "67 = 55?", "92 = 55?", e assim por diante. Após seis comparações, encontramos o valor 55 no quadro. O valor 55 pode ser visto como o identificador de um cliente. Ao recuperar a estrutura que está armazenada na posição onde o 55 está, estão armazenadas todas as informações de seu cadastro, como nome, telefone, *e-mail*, entre outros.

Agora, vamos buscar o número 20 no mesmo quadro. Da esquerda para a direita, a busca sequencial vai comparando elemento a elemento em busca do 20. Os testes começam com "67 = 20", "92 = 20", até o final do quadro, onde o teste "13 = 20" é feito. Como chegamos ao final do quadro sem encontrar o elemento 20, o método retornaria uma mensagem tal como "O elemento 20 não foi encontrado".

O algoritmo que implementa a técnica de busca sequencial pode ser utilizado para buscar e recuperar dados em diversas estruturas de dados, como, por exemplo, listas e vetores. Para tanto, é necessário passar como parâmetro para o algoritmo um vetor ou lista e a chave do elemento que se deseja encontrar.

Link

Acesse o *link* a seguir para conhecer melhor a busca sequencial.

https://qrgo.page.link/qnAJ2

O pseudocódigo desse algoritmo pode ser visto no quadro a seguir. Na linha 1, utilizando um laço, percorremos da posição 1 até a N de uma lista ou vetor. Na linha 2, realizamos um teste para verificar se a chave buscada é a mesma do elemento na posição `i` do vetor. Caso esse teste seja verdadeiro, retornamos o elemento `i`, caso contrário, incrementamos o contador `i` e continuamos a execução. Ao final, caso não seja encontrado, retorna a mensagem "chave não encontrada".

Pseudocódigo	Implementação em Python

BUSCA-SEQUENCIAL[A, chave, N]
1. **para** i = 1 **até** N
2. **se** A[i] = chave **então**
3. **retorna** A[i]
4. i = i + 1
5. **retorna** "chave não encontrada"
	`def busca_sequencial(A, chave, N):`
	`  for i in range (0, N):`
	`  if(A[i] == chave):`
	`  return A[i];`
	`  return "chave não encontrada"`

A busca sequencial é indicada para pequenos conjuntos de dados, como, por exemplo, listas e vetores de até 100 elementos. Para tamanhos maiores, outras técnicas, como a busca binária e a tabela *hash*, apresentam mais vantagens em relação ao desempenho. A complexidade de espaço do algoritmo é O(1) e a de tempo varia de acordo com o caso. No melhor caso, o elemento buscado é o primeiro do vetor e, nesse caso, a complexidade é O(1). No pior caso, o elemento não existe e iremos percorrer todo o vetor mesmo assim. Nesse caso, a complexidade é O(N).

A busca binária é uma técnica de pesquisa eficiente. Sua única restrição é que as chaves do conjunto de dados devem estar ordenadas. Logo, existe um custo inicial para armazenar os dados já ordenados ou executar um algoritmo de ordenação antes da busca binária.

Esse método segue um padrão de projeto de algoritmos chamado divisão e conquista. A divisão ocorre nos acessos aos dados. Diferentemente da busca sequencial, que acessa elementos da esquerda para direita, do início ao fim do vetor, a busca binária faz acessos à posição do meio do vetor comparando esse valor com a chave e, após, descartando uma das metades restantes. A cada iteração, o local de busca no vetor diminui pela metade, assim, tendo um bom desempenho.

Fique atento

Na busca binária, as chaves devem estar ordenadas, mas não necessariamente os dados. Uma lista pode ter como chave o código do aluno. Esses códigos devem estar em ordem crescente ou decrescente no vetor, mas os dados — por exemplo, o nome do aluno — não precisam estar ordenados.

Por exemplo, em uma base de dados armazenada em arquivo, temos uma lista de cursos e suas informações. Uma vez que são poucos cursos, eles estão armazenados em um arquivo de texto ao invés de um banco de dados SQL. Foi requisito para nós o desenvolvimento de uma função que busca cursos pelo nome e retorna as suas informações. Após analisar os requisitos, decidimos utilizar a busca binária. A lista de cursos encontra-se no quadro a seguir.

Ciência da computação	Engenharia de energia	Geologia	Jornalismo	Matemática	Nutrição	Pedagogia

O primeiro passo é verificar se a lista está ordenada. Caso sim, continuamos, caso contrário, executamos um algoritmo de ordenação. Verificamos que a lista de sete cursos está ordenada por nome, em ordem ascendente. Na busca binária, em cada iteração, acessamos a posição do meio do vetor ou lista. No primeiro exemplo, vamos buscar o curso Nutrição. O primeiro teste será "Jornalismo = Nutrição"? A resposta será não. Uma vez que a letra N vem após a letra J, podemos ignorar todo o lado esquerdo do vetor a partir de Jornalismo. A próxima iteração vai considerar apenas o vetor que começa em Matemática e vai até o fim. Esse vetor tem três elementos, sendo que o do meio é Nutrição. No segundo teste, "Nutrição = Nutrição"?, encontramos o curso buscado.

O segundo exemplo tem como objetivo mostrar o que acontece quando o elemento não existe no vetor. Para tanto, vamos buscar o curso Direito. Na primeira iteração, vamos comparar "Jornalismo = Direito"? A resposta será não. Levando-se em consideração que D vem antes de J, vamos ignorar todo o vetor de Jornalismo até o fim. Agora, vamos processar o vetor que começa em Ciência da computação e vai até Geologia. Na segunda iteração,

o teste feito será "Engenharia de energia = Direito"? Novamente a resposta será não, e tudo de Engenharia de energia até o fim será ignorado. Por fim, na terceira iteração, o vetor, agora de uma posição, contém apenas o curso Ciência da computação. O teste "Ciência da computação = Direito" será feito e uma vez que não existem outras posições do vetor para procurar, o algoritmo retornaria a mensagem "Chave não encontrada".

O pseudocódigo desse algoritmo pode ser visto no quadro a seguir. Nas linhas 1 e 2, iniciamos os intervalos esquerda e direita. Na linha 3, um laço é feito para repetir o algoritmo até que o intervalo acabe. Após, na linha 4, calculamos o meio e, na linha 5, começamos os testes. Se o elemento do meio for o que está sendo procurado, retornamos ele. Caso não seja verdade, na linha 7, verificamos qual metade do vetor podemos ignorar: a metade da esquerda, se o valor que buscamos é maior que o meio, ou a metade da direita, caso contrário. Ao final, caso não encontre, retorna a mensagem "chave não encontrada".

Pseudocódigo	Implementação em Python

```
BUSCA-BINÁRIA[A, chave, N]
1.  esquerda = 1
2.  direita = N
3.  enquanto esquerda <= direita
4.      meio = (esquerda + direita) / 2
5.      se A[meio] = chave então
6.          retorna A[meio]
7.      senão se A[meio] < chave então
8.          esquerda = meio + 1
9.      senão
10.         direita = meio - 1
11. retorna "chave não encontrada"
```

```
1.  def busca_binaria(A, chave, N):
2.      esquerda = 0
3.      direita = N - 1
4.      while(esquerda <= direita):
5.          meio = int((esquerda +
    direita) / 2)
6.          if A[meio] == chave:
7.              return A[meio]
8.          elif A[meio] < chave:
9.              esquerda = meio + 1
10.         else:
11.             direita = meio - 1
12.     return "chave não encontrada"
```

A busca binária é indicada para médios e grandes conjuntos de dados, como, por exemplo, listas e vetores a partir de 100 elementos. Para tamanhos menores, a busca sequencial é suficiente. A complexidade de espaço do algoritmo é O(1) e a de tempo varia de acordo com o caso. No melhor caso, o elemento buscado é o primeiro do vetor e, nesse caso, a complexidade é O(1). No pior caso, o elemento não existe e iremos percorrer O(`log_2 N`) posições. Isso ocorre, pois, em cada iteração, cortamos pela metade o espaço de exploração.

Saiba mais

Para entender melhor o conteúdo deste capítulo, leia a obra *Estruturas de dados: volume 18*, de Nina Edelweiss e Renata Galante (2009).

2 Tabelas *hash* e suas aplicações

A busca sequencial e binária procura informações armazenadas com base na comparação de suas chaves. Por exemplo, busca um cliente por nome em um vetor de clientes. A solução eficiente, que é a busca binária, tem como restrição a ordenação dos dados.

A tabela *hash* é uma estrutura de dados especial que associa chaves de pesquisa a valores para, assim, fazer buscas rápidas e obter os valores desejados. Segundo Mailund (2019), tabelas *hash*, quando implementadas corretamente, são as estruturas mais eficientes para representar conjuntos de dados e pesquisas.

Diversas linguagens de programação, como C++ e Python, utilizam tabelas *hash* para a implementação de dicionários e *sets*. Alguns dos serviços mais utilizados no mundo, como o DNS (Domain Name System) e o DHCP (Dynamic Host Configuration Protocol), também utilizam essa estrutura de dados.

Link

Acesse o *link* a seguir para entender a importância de *hash* em segurança da informação.

https://qrgo.page.link/SfhQj

A ideia central de tabelas *hash* é utilizar uma função (*hash*) aplicada sobre uma chave para retornar o índice no qual a informação está armazenada. A função *hash* mapeia uma chave para um índice único. No exemplo a seguir, vamos verificar como são armazenados os dados em uma tabela *hash*.

Exemplo

Vamos assumir que um sistema está sendo construído para administrar uma universidade. Você é o responsável por implementar a parte do sistema que pesquisa alunos e retorna suas informações, como curso, ano de ingresso, entre outros.

Uma vez que várias buscas serão feitas ao longo da vida útil do sistema, você decide utilizar uma tabela *hash* para armazenar os dados dos alunos. A tabela tem o seguinte formato: nome, curso e ano de ingresso. Confira a seguir um exemplo dessa tabela.

[1] – Camila; Direito; 2012;
[2]
[3]
[4] – Gabriela; Psicologia; 2018;
[5]
[6]
[7] – Bruna; Fisioterapia; 2019;
[8]
[9] – Sarah; Nutrição; 2012;

Dada a tabela, uma função *hash* deve ser escolhida buscando converter as chaves (nomes) em índices do vetor ou lista.

Tabelas *hash* podem ser implementadas utilizando vetores de listas. Cada posição do vetor guarda uma lista de elementos. Se a função *hash* escolhida for boa, poucas colisões vão ocorrer. Colisões ocorrem quando duas ou mais chaves geram o mesmo índice da tabela *hash*. E são comuns, pois, em geral, o número de chaves é muito maior que o tamanho da tabela.

Quando as colisões ocorrerem, adicionamos mais um elemento na lista encadeada. Isso acarreta que, na busca, caso existam muitas colisões no índice da chave buscada, uma lista encadeada vai precisar ser percorrida e o desempenho será perdido.

A função *hash* pode mapear, por exemplo, chaves que são *strings* para índices que são inteiros. Um exemplo de função *hash* para caracteres é a que considera o código ASCII do primeiro caractere de uma *string* para definir o índice da tabela em que aquela chave será armazenada. Também existem funções *hash* que multiplicam o valor dos códigos ASCII de cada caractere por um número primo, buscando gerar um índice diferente para cada chave.

O mesmo ocorre para chaves que são números inteiros, entretanto, nesse caso, não é necessário considerar o código ASCII, e sim o próprio valor. Multiplicações com números primos são muito comuns, pois geram distribuições mais uniformes. Após o cálculo do índice, o qual pode gerar um valor maior que o tamanho da tabela, calculamos o resto da divisão pelo tamanho da tabela para, assim, gerar um índice que vai de 1 até o tamanho da tabela.

Link

Funções *hash* também são utilizadas para criptografia de dados e chaves de acesso a servidores. Existe uma classe extensa de funções *hash* para isso. Leia mais sobre o assunto no *link* a seguir.

https://qrgo.page.link/omjyw

A implementação de tabelas *hash* requer um bom conhecimento sobre outras estruturas de dados. O pseudocódigo, no quadro a seguir, mostra um exemplo de função *hash* para *strings*. Na linha 1, pegamos o primeiro caractere da *string* em ASCII e calculamos o resto da divisão para gerar um índice no intervalo permitido pela tabela *hash*. Na linha 2, retornamos o valor do índice.

Pseudocódigo	Implementação em Python

FUNÇÃO-HASH[chave]
1. valor = chave[0] % TAMANHO_TABELA
2. **retorna** valor
	`def funcao_hash(chave):`
	`  valor = chave % len(tabela_hash)`
	`  return valor`

Para inserir e alterar um valor, podemos utilizar a mesma função. O pseudocódigo do quadro a seguir implementa essa funcionalidade. Na linha 1, descobrimos a posição do vetor na qual a chave deve ser armazenada. Na linha 2, recuperamos a lista na qual a chave deve estar. Após, na linha 3, percorremos a lista encadeada (pois podem ter ocorrido colisões) em busca da chave. Na linha 4, testamos se a chave existe. Caso exista, alteramos o valor na linha 5 e, caso não exista, na linha 6, continuamos buscando por ela na lista. Por fim, caso a chave não seja encontrada, na linha 7, criamos um elemento e adicionamos a chave e o valor, respectivamente.

Pseudocódigo	Implementação em Python

INSERE-ALTERA-HASH[tabela, chave, valor]
1. posição = FUNCAO-HASH(chave)
2. entrada = tabela[posição]
3. **enquanto** entrada != nulo
4. **se** entrada.chave == chave **então**
5. entrada.valor = valor
6. entrada = entrada.proximo
7. entrada = {chave, valor}
	`def insere(tabela_hash, chave, valor):`
	`  linha = funcao_hash(chave)`
	`  chave_existe = False`
	`  entrada = tabela_hash[linha]`
	`  for i, chave_valor in enumerate(entrada):`
	`  chave_atual, valor_atual = chave_valor`
	`  if chave == chave_atual:`
	`  chave_existe = True`
	`  break`
	`  if chave_existe:`
	`  entrada[i] = ((chave, valor))`
	`  else:`
	`  entrada.append((chave, valor))`

Outra função importante é a de obter o valor dado a uma chave. O pseudocódigo a seguir implementa essa funcionalidade. A implementação é semelhante com a função de inserir. Nas linhas 1 e 2, utilizamos a função *hash* para descobrir a posição do vetor no qual a chave deveria estar e, após, recuperamos a lista encadeada onde ela possivelmente se encontra. Após, na linha 3, percorremos essa lista e, na linha 4, testamos se a posição atual da lista é a da chave buscada. Na linha 5, caso o resultado do teste seja verdadeiro, retornamos o valor. Caso falso, na linha 6, avançamos na lista buscando a próxima posição. Por fim, na linha 7, caso não seja encontrada a chave, retornamos uma mensagem informando esse fato.

Saiba mais

As vantagens das tabelas *hash* são a velocidade da busca. A complexidade de tempo da busca no caso médio é O(1) e a desvantagem vem do fato que quando escolhemos uma função *hash* pouco eficiente, muitas colisões ocorrem e, nesse caso, a complexidade de tempo passa a ser O(n), onde n é o número de elementos que colidiram naquela posição da tabela. Leia mais sobre as vantagens e as desvantagens no *link* a seguir.

https://qrgo.page.link/gDXyh

3 Aplicação dos métodos de busca em Python

Os métodos de busca sequencial e binária têm diversas aplicações tanto em sistemas de gestão quanto na implementação de ferramentas e do sistema operacional em si. A seguir, vamos apresentar exemplos de desenvolvimento de problemas propostos, utilizando os métodos de pesquisa em lista vistos nas seções anteriores e a respectiva solução deles.

Os exemplos foram todos implementados em linguagem de programação Python, versão 3. Nossos exemplos contêm tamanhos de entrada pequenos, mas assumem que, em um sistema em produção, o número de elementos de uma lista ou vetor pode ultrapassar os milhões e até os bilhões de elementos.

Exemplo

Para o primeiro exemplo, assuma que um cliente solicitou a construção de um sistema para a gestão de uma universidade. Você está construindo um método no qual o nome do aluno é passado via parâmetro e o curso dele deve ser obtido. Os dados dos alunos estão sendo armazenados em um vetor A de tamanho N posições. O cliente também informa que o número de alunos da universidade é pequeno, com menos de 100 alunos no total. Qual método de busca você sugere?

Uma das possíveis respostas é a utilização da busca sequencial. No exemplo a seguir, mostramos uma implementação desse problema em Python. Na linha 9, declaramos um vetor de pares Aluno e Curso. Na linha 11, pedimos ao usuário que digite o nome do aluno que se deseja pesquisar. Após, na linha 12, obtemos o tamanho do vetor e, na 14, invocamos a função que implementa a busca sequencial. Por fim, na linha 15, mostramos o resultado na tela, o qual é o nome do curso ou uma mensagem informando que o aluno não foi encontrado.

A implementação da busca sequencial começa na linha 1 com a sua declaração. Na linha 3, utilizando um laço de repetição *for*, percorremos o vetor da posição 0 até a posição N – 1. Após, na linha 4, testamos se o nome do aluno na posição `i` é o mesmo nome que estamos buscando. Caso verdade, retornamos o nome do curso, caso contrário, continuamos a execução. Por fim, na linha 6, caso o aluno não seja encontrado, retornamos uma mensagem de não encontrado.

```
1.  def busca_sequencial(A, chave, N):
2.
3.      for i in range (0, N):
4.          if (A[i][0] == chave):
5.              return A[i][1];
6.      return "Aluno não encontrado"
7.
8.
9.  A = [["Matheus", "Ciência da
    Computação"], ["Arthur", "Administraçao"], ["Juliana", "Engenharia
    Elétrica"], ["Carolina", "Letras"], ["Luana", "Relações Públicas"]];
10.
11. chave = input("Digite o nome de um aluno para pesquisa: ")
12. N = len(A);
13.
14. resultado = busca_sequencial(A, chave, N)
15. print(resultado)
```

O próximo exemplo trata do mesmo problema, entretanto, o sistema encontra-se em produção e o número de alunos vem crescendo ano após ano. O cliente informou que a busca demora muito tempo para retornar o nome do curso e agora o requisito dele é mais desempenho nesse método.

Nesse caso, uma das possíveis soluções é a implementação de um método de busca eficiente, como a busca binária. Lembre-se que a única restrição desse método é que os valores das chaves devem estar ordenados no vetor. Nesse sentido, ordenamos dentro da função ou inserimos os dados sempre ordenados para garantir essa restrição.

Exemplo

O exemplo a seguir implementa uma solução para esse problema utilizando a linguagem de programação Python 3. Na linha 17, declaramos um vetor de pares Aluno e Curso. Na linha 19, pedimos ao usuário que digite o nome do aluno que se deseja pesquisar. Após, na linha 20, obtemos o tamanho do vetor e, na 22, invocamos a função que implementa a busca binária. Por fim, na linha 23, mostramos o resultado na tela, o qual é o nome do curso ou uma mensagem informando que o aluno não foi encontrado.

A implementação da busca binária começa na linha 1 com a sua declaração. Nas linhas 2 e 3, definimos o intervalo que, na primeira iteração, vai da posição 0 até a `N – 1`. Na linha 5, utilizando um laço de repetição *while*, repetimos a busca até encontrar o valor ou até o intervalo de busca terminar. Isso ocorre quando o intervalo da direita é menor que o da esquerda. Após, na linha 6, calculamos a posição do meio do inter-

valo, a qual será verificada se é a chave buscada ou não. Na linha 7, realizamos essa verificação e, caso seja a chave correta, retornamos o nome do curso. Caso contrário, na linha 9, verificamos qual metade do vetor ou lista devemos ignorar. Para decidir isso, comparamos o valor que buscamos com o valor do meio. Uma vez que o vetor está ordenado, podemos ignorar uma de suas metades. Esse procedimento se repete até que encontramos ou não a chave e, logo, retornamos uma mensagem de aluno não encontrado.

```
1.  def busca_binaria(A, chave, N):
2.      esquerda = 0
3.      direita = N - 1
4.
5.      while(esquerda <= direita):
6.          meio = int((esquerda + direita) / 2)
7.          if A[meio][0] == chave:
8.              return A[meio][1]
9.          elif A[meio][0] < chave:
10.             esquerda = meio + 1
11.         else:
12.             direita = meio - 1
13.
14.     return "Aluno não encontrado"
15.
16.
17. A = [["Arthur", "Administraçao"], ["Carolina", "Letras"], ["Juliana", "Engenharia
    Elétrica"], ["Luana", "Relações Públicas"], ["Matheus", "Ciência da Computação"]];
18.
19. chave = input("Digite o nome de um aluno para pesquisa: ")
20. N = len(A);
21.
22. resultado = busca_binaria(A, chave, N)
23. print(resultado)
```

Nosso terceiro exemplo trata de vetores de valores inteiros, e não de *strings*, como nos dois exemplos anteriores. Um cliente, dono de um prédio de estacionamento, solicitou o desenvolvimento de um sistema para a gestão de seu negócio. O módulo que você irá implementar trata de verificar se determinada vaga está livre ou não. O cliente tem um prédio de 10 andares, sendo que cada andar contém 40 vagas.

Exemplo

O exemplo a seguir implementa uma solução para esse problema utilizando a linguagem de programação Python. Na linha 17, temos a declaração do vetor de vagas. Cada um dos valores já ordenados indica quais são os identificadores das vagas livres no momento. Após, na linha 19, solicitamos ao usuário qual vaga ele deseja verificar.

Na linha 20, por meio da função *len*, retornamos o número de elementos desse vetor. Na linha 22, o programa chama a função de busca, a qual retorna −1, caso a vaga esteja livre, e 1, caso esteja ocupada. Na linha 24, testamos se o resultado foi −1, neste caso, informamos que a vaga está disponível, caso contrário, informamos que ela está ocupada.

A implementação da busca binária começa na linha 1. Inicialmente, nas linhas 2 e 3, definimos o intervalo esquerda e direita. Após, na linha 5, utilizando um laço de repetição *while*, percorremos o espaço de busca até que não existam mais posições para testar. Na linha 6, calculamos a posição do elemento do meio do intervalo atual. E, na linha 7, testamos se o valor é a chave que estamos buscando. Caso verdade, retornamos 1, informando que a vaga está ocupada, caso contrário, continuamos nas linhas 9 e 11 testando qual metade do vetor podemos ignorar. Por fim, caso o elemento não seja encontrado após a busca, retornamos −1, informando que a vaga está livre.

```
1.  def busca_binaria(A, chave, N):
2.      esquerda = 0
3.      direita = N - 1
4.
5.      while(esquerda <= direita):
6.          meio = int((esquerda + direita) / 2)
7.          if A[meio] == chave:
8.              return 1
9.          elif A[meio] < chave:
10.             esquerda = meio + 1
11.         else:
12.             direita = meio - 1
13.
14.     return -1
15.
16.
17. A = [16, 20, 25, 27, 31, 46, 76, 78, 82, 98];
18.
19. chave = int(input("Digite um número de vaga para testar: "))
20. N = len(A);
21.
22. resultado = busca_binaria(A, chave, N)
23.
24. if resultado != -1:
25.     print("Vaga disponível")
26. else:
27.     print("Vaga ocupada")
```

Neste capítulo, estudamos os métodos de pesquisa em vetores e listas, a estrutura de dados em uma tabela *hash* e as aplicações desses em diferentes situações. Além disso, compreendemos as vantagens e as desvantagens de cada um, visualizando exemplos de implementação em linguagem de programação Python.

Referências

DESJARDINS, J. *How much data is generated each day?* 2019. Disponível em: https://www.visualcapitalist.com/how-much-data-is-generated-each-day/. Acesso em: 16 fev. 2020.

MAILUND, T. *The joys of hashing*: hash table programming with C. New York: Apress, 2019.

Leitura recomendada

EDELWEISS, N.; GALANTE, R. M. *Estruturas de dados*. Porto Alegre: Bookman, 2009.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS