PESQUISA, ORDENAÇÃO E TÉCNICAS DE ARMAZENAMENTO

UNIDADE 1 – CONCEITUAÇÃO DE PESQUISA, ORDENAÇÃO E ARMAZENAMENTO

Luiz Ricardo Mantovani da Silva

Introdução

O volume de dados sempre foi um desafio na área de desenvolvimento de software. De um lado, os requisitos de usuários exigem que as operações sejam mais detalhadas, mais dados sejam coletados e armazenados e que mais análises sejam realizadas sobre esses dados. Por outro lado, os dispositivos vem sendo reduzidos. Quais tecnologias poderiam descrever este cenário? Celulares e tablets são exemplos claros de que o processamento de dados precisará ser profundamente estudado, pois, embora sua capacidade seja grande se comparada com equipamentos do passado, esses dispositivos não possuem a mesma capacidade que os computadores atuais. E quando a situação exige desempenho e discrepância? Para enriquecer ainda mais o cenário, o uso de dispositivos vestíveis, smart watches, e dispositivos embarcados é frequente e requer que os desenvolvedores de sistemas sejam hábeis na solução de problemas de ocupação de espaço e velocidade de processamento.

Como fazer quando o equipamento viabiliza apenas a utilização de poucos recursos computacionais? Considere, por exemplo, o sistema de controle de deslocamento de uma bicicleta ou patinete compartilhados. Não há condições de se embarcar um dispositivo com grande capacidade de processamento, mas, mesmo assim, é necessário liberar e bloquear o veículo, apontar, via geoposicionamento, onde está, registrar o deslocamento, saber que o usuário está liberado, saber que o veículo está bloqueado. São inúmeras operações a serem executadas na bicicleta ou no patinete mas com recursos computacionais bastante limitados.

Se, porém, levarmos em conta somente os computadores convencionais (PCs e notebooks), mesmo com mais recursos, o volume de dados processado é cada vez maior. Portanto, o desenvolvedor do software precisa estar atento às melhores técnicas para armazenamento, busca e ordenação de dados, sendo capaz de aplicá-las para a construção de funcionalidades altamente eficientes. Vamos lá?

1.1 Conceitos sobre Pesquisa, Ordenação e Formas de Armazenamento das Informações

Pesquisar dados é de forma geral, uma busca por uma chave dentro de uma estrutura de armazenamento. Essa estrutura, por sua vez, precisa estar minimamente organizada, ordenada portanto, para garantir a eficiência da busca.

O primeiro assunto a tratarmos é a pesquisa ou busca propriamente dita; imagine a seguinte situação: queremos encontrar um amigo dentro de uma festa; para facilitar a sua localização, é interessante que conheçamos os hábitos dele. Certo? É isto que acontece com os dados, quando tentamos localizá-lo em meio a milhões de informações. A tarefa pode ficar difícil e custosa do ponto de vista do processamento, no entanto, se aplicarmos métodos para localizar os dados que queremos, poderemos encontrá-los em um tempo menor e com uma carga de processamento reduzida. Esta é a tarefa dos algorítimos de busca, muito utilizados no mundo da programação. (PUGA; RISSETTI, 2004).

A ordenação dos dados é outra importante questão que iremos tratar: ordenar significa mudar a posição dos elementos de uma lista, colocando-os em uma determinada ordem. Desta maneira, os dados ordenados são mais fáceis de se localizar, por exemplo, a pesquisa em uma base de dados ordenada é muito mais rápida. Os algoritmos para ordenação de dados são as ferramentas utilizadas para organizar as informações; o melhor método a ser utilizado depende muito do problema apresentado, devendo ser adequado da melhor forma possível. De todos os métodos disponíveis, o da bolha é considerado o modelo de ordenação mais simples que existe, porém, pode ser utilizado em certos tipos de dados.

A busca e ordenação dos dados está intimamente ligado ao desempenho computacional, mas outra relevante questão, senão a mais importante é o armazenamento dos dados.

1.1.1 Armazenamento de dados

Armazenar dados é reter os dados no computador, de forma que sejam utilizados em processos, mediante sua recuperação e uso. Há duas formais gerais para armazenamento de dados. Armazenamento interno, que trata de manter os dados dentro das instâncias do programa que está sendo executado (memória) ou armazenamento externo, que trata de gravar os dados em objetos externos ao programa, sendo executado como arquivos em disco ou bancos de dados.

- **Armazenamento interno**

O armazenamento interno consiste em manter os dados na memória do programa que está em execução. Assim, cada instrução do programa pode acessar tais dados e processá-los de acordo com a lógica necessária. A estrutura mais elementar para armazenamento de dados é uma variável de memória, que consiste em um espaço de memória reservado para receber um determinado dado; recebe um nome para referência, o qual será utilizado para processá-la. (MANZANO; OLIVEIRA, 2016).

No entanto, há situações em que as variáveis são insuficientes. Por exemplo, um programa que recebe três valores e os imprime ordenados poderia ser assim escrito:

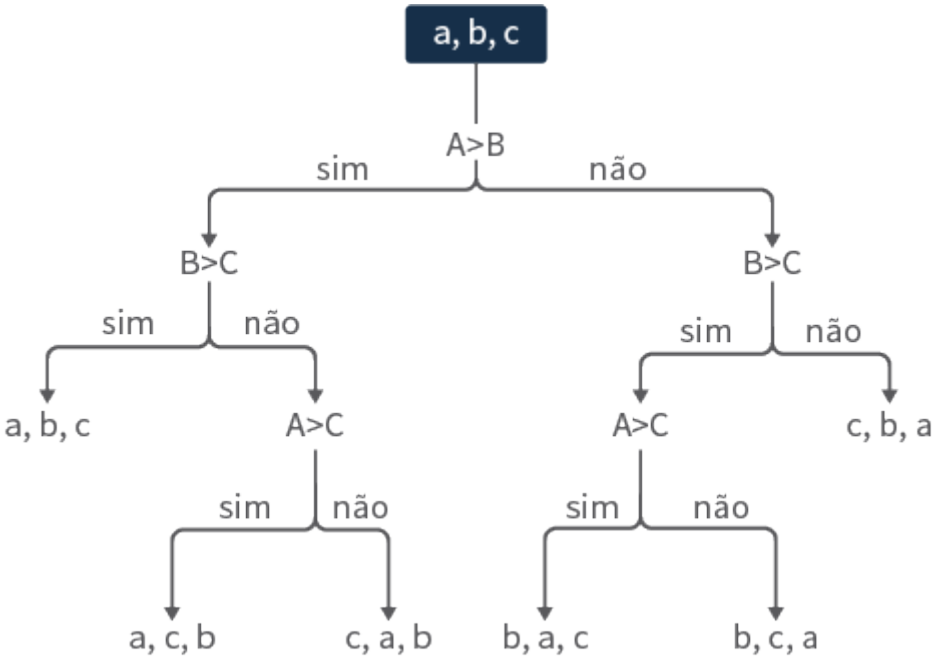

Figura 1 - Fluxograma para ordenar três variáveis.
Fonte: Elaborada pelo autor, 2019.

Perceba que seis possíveis resultados seriam escritos no programa em questão, ou melhor, três resultados. No entanto, no caso de o programa precisar ordenar 12 valores, seriam escritos 12 resultados diferentes. Ou seja, 479.001.600 resultados. Além disso, seria difícil programar com dez nomes diferentes de variáveis, pois teríamos que implementar grande quantidade de estruturas condicionais para verificar os valores. (SOUZA et al, 2011). Precisamos de uma forma mais eficiente de tratar dados em conjunto. Para isso, existem os vetores, que são variáveis indexadas e permitem armazenar mais de um dado em uma estrutura.

Vetores (variáveis indexadas unidimesionais)

Ao observar um prédio, pode-se ver os andares, referenciados por seus números: primeiro andar, segundo andar e assim, sucessivamente. Dentro de cada andar, também encontramos salas, que, por sua vez, também podem ser

nomeadas por números: sala um, sala dois, sala três etc. Esse é o conceito de variável indexada unidimensional: tem-se um conjunto de elementos do mesmo tipo que são organizados conforme um só nome; no caso do nosso exemplo, os andares ou salas; e que são acessados por um número, índice.

Figura 2 - Representação dos vetores prédio e andar.
Fonte: Elaborada pelo autor, 2019.

Se variarmos o número, poderemos acessar a cada uma das salas dentro do vetor Salas, como no algoritmo (1).

```
Para i de 1 até 4 passo 1 faça
escreva Salas[i]
Fim_para
```

(Algoritmo 1 - fragmento algoritmo: um laço de repetição para navegar no vetor Salas)

Por convenção, as variáveis indexadas unidimensionais são chamadas de vetores, e podem ser de qualquer tipo de dados válido, como textos ou inteiros. Sua implementação se dá, em Portugol, da seguinte maneira:

```
Inteiro vetor_n[10] // Criação de um vetor de 10 inteiros
vetor_n = {10,20,30,40,50,60,70,80,90,100} //Atribuir 10 valores para o vetor
```

(Algoritmo 2 - fragmento: implementação de um vetor de inteiros com 10 elementos)

Afirma-se que o vetor_n é unidimensional por que possui uma sequencia de elementos, alinhadas entre si no mesmo nível, assim, para acessar qualquer dos elementos, basta invocar o vetor_n a partir de um índice válido. Por exemplo:

```
Escreva vetor_n[3]
```

Resultado: 30

(Algoritmo 3 - fragmento: acesso ao 3º elemento do vetor_n)

Assim, um vetor é capaz de armazenar blocos de dados, mas com a restrição de que tais dados serão sempre do mesmo tipo. Um vetor de *strings* não poderá conter valores inteiros, por exemplo.

VOCÊ SABIA?

Iterar significa navegar em uma estrutura de dados. Para vetores, por exemplo, se trata de criar um laço de repetição que nos permita capturar cada elemento v[i], onde i é o índice sendo iterado. A estrutura é constituída basicamente de um vetor v[i] onde v representa a variável e o [i] a variável índice do vetor, que por exemplo, é incrementada pelo laço de repetição, assim, toda vez que o trecho do algoritmo dentro do laço é executado, o valor de i é alterado e o v[i] recebe um novo valor, ex.: v[0], v[1], v[2], v[3]....

Os vetores são estruturas de dados que servem para economia de código, ou seja, uma forma de se atribuir a uma única variável diversos valores, simplesmente utilizando um índice durante um processo de iterações. Esta estrutura é muito utilizada em programação de computadores, contribuindo para redução de código e aumento de eficiência.

VAMOS PRATICAR?

Implemente um programa que receba 10 valores em um vetor de inteiros. Após isso, exiba a soma desses valores, a média, o menor valor informado e o maior valor informado.

Resolução

Pseudocódigo:

Var // Comando que indica onde as variáveis serão declaradas.

Valores: vetor [1..10] de reais // Declaração da variável Valores do tipo vetor de 10 posições.

Soma, Media : real // Declaração das variáveis Soma e Media do tipo real.

i, maior, menor: inteiro // Declaração das variáveis i, maior e menor do tipo inteiro.

Início // Comando que indica onde começam as instruções.

Soma <-- 0 // Variável recebendo valor inicial.

Para i <-- 0 até 9 faça // Início do laço de repetição.

Ler (Valores[i]) // Comando para leitura de valores digitados no teclado.

Soma <-- Soma + Valores[i] // Variável Soma recebendo o valor da operação aritmética.

Se (Valores[i]<menor) Então // Estrutura condicional que realiza uma comparação, verificando se a variável Valores[i] é < que a variável menor.

menor <-- Valores[i] // Caso a condição seja verdadeira, variável menor recebendo o valor da variável Valores [i].

Fim- Se // Fim da estrutura condicional.

Se (Valores[i]>maior) Então // Estrutura condicional que realiza uma comparação, verificando se a variável Valores[i] é > que a variável maior.

maior <-- Valores[i] // Caso a condição seja verdadeira, variável maior recebendo o valor da variável Valores [i].

Fim- Se // Fim da estrutura condicional.

Fim – Para // Fim da estrutura de repetição.

Media <-- Soma/10 // Variável Media recebendo o resultado da média (resultado de Soma/ 10).

Mostrar (“O valor da média é:”, Media) // Comando que exibe o valor da variável Média.

Mostrar (“O valor da soma é:”, Soma) // Comando que exibe o valor da variável Soma.

Mostrar (“O maior valor é:”, maior) // Comando que exibe o maior valor informado.

Mostrar (“O menor valor é:”, menor) // Comando que exibe o menor valor informado.

Fim // Fim do programa.

Matrizes (vetores bidimensionais)

É possível armazenar os dados em estruturas bidimensionais, com linhas e colunas, na forma de uma tabela de dados. Essas estruturas são, na verdade, vetores, mas com uma dimensão adicional. Essas estruturas também são chamadas de matrizes.

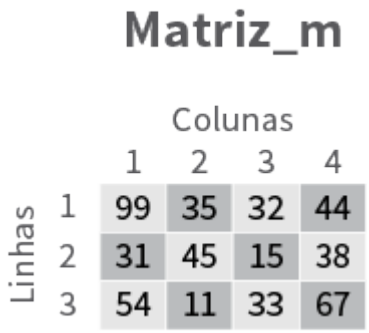

Figura 3 - Representação de uma matriz.
Fonte: Elaborada pelo autor, 2019.

Assim como nos vetores unidimensionais, o acesso aos elementos se dará mediante índices, mas, no caso, a partir da coordenada (linha, coluna). Por exemplo:

Escrever Matriz_M[2,3] //Imprimir o conteúdo do elemento linha 2, coluna 3

Resultado: 15

(Algoritmo 4 - fragmento: acesso a um determinado elemento da Matriz_m)

Para interagir em uma matriz, implementam-se dois contadores, uma para as linhas e outro para as colunas, mediante laços aninhados (um laço dentro do outro). Por exemplo:

inteiro Matriz_m[3][4]
... //bloco para carregar dados na matriz
Para linha de 1 até 3, passo 1, faça
Para coluna de 1 até 4, passo 1, faça
Escrever Matriz_m[linha][coluna] //Imprime o valor de cada célula
Fim_para
Fim_para

(Algoritmo 4 - fragmento: acesso aos elementos de uma matriz com laços aninhados)

De forma geral, os vetores têm funcionalidade similar a tabelas, armazenado os dados em linhas e colunas e permitindo sua leitura mediante coordenadas, os índices dos elementos. Por isso, durante suas pesquisas, você encontrará autores que se referirão a vetores como tabelas, uma analogia adequada. Em memória, os vetores são representados de forma similar às variáveis, mas, cuja referência está indexada e aponta para mais de um espaço de memória.

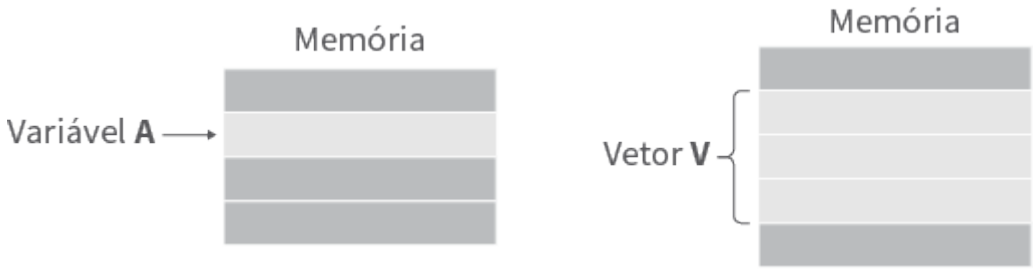

Figura 4 - Esquema de locação de memória em variáveis e vetores.
Fonte: Elaborada pelo autor, baseada em SOUZA et al, 2011.

Hash Table

Uma tabela *Hash* - *Hashtable* - é uma estrutura que permite a rápida recuperação de dados indiferentemente do volume de dados armazenados nela. Por essa razão, tabelas *hash* são amplamente utilizadas em índices de banco de dados, compiladores e sistemas de registro (*log*) de erros. Diferente dos vetores e matrizes, a tabela *hash* armazena dados organizando-os em pares de chave-valor (*key*,*value*). Em cada elemento, pode-se armazenar, por exemplo, o nome de uma pessoa e seu número de telefone; essas estruturas são chamadas entradas (*entries*). (GOOGDRICH; TAMASSIA, 2013).

VOCÊ QUER VER?

Você tem dúvidas sobre vetores e matrizes e como são utilizados na computação? No vídeo do Grupo de Estudos de Educação à Distância, do Centro Paula Souza , você poderá aprender mais sobre estas estruturas de dados de uma forma didática e com exemplos práticos. Assista a este vídeo clicando aqui <https://www.youtube.com/watch?v=7i3YfiLy1vE&list=LLecA4jNMub1jeYn7rRe__gA&index=2&t=0s>.

Se consideramos um vetor, ilustrado na figura a seguir, para localizar um determinado dado (Ana), pode-se utilizar uma estratégia de força bruta, percorrendo cada um dos elementos até que se encontre o elemento com a informação desejada. Quanto maior o vetor, mais tempo será necessário para localizar o dado.

Texto nomes[11] = {“Jane”, “Tim”, “Mia”,”Sam", “Leo”, “Ted”, “Bea”, “Lou”, “Ana”, “Max”, "Zoe"}
Para i de 0 até 10, passo 1, faça
se nomes[i] = “Ana" então
escreve i
fim_para
fim_se
Fim_para

Algoritmo 5 - fragmento: buscar por “Ana" no vetor nomes mediante força bruta.

Figura 5 - Busca utilizando força bruta.
Fonte: Elaborada pelo autor, 2019.

Porém, se o índice do elemento fosse previamente conhecido, acessaríamos diretamente o elemento, sem percorrer os demais.

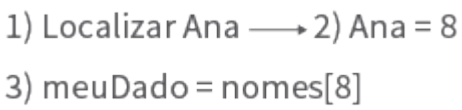

Figura 6 - Esquema lógica para localizar dados dentro de uma tabela hash.
Fonte: Elaborada pelo autor, 2019.

Função *hash*

Ocorre que, em uma tabela *hash*, o índice dos dados é determinado mediante um cálculo com os dados. O que significa dizer que o índice do elemento está relacionado com os dados armazenados. Esse procedimento de determinação do índice relativo à chave é executado por uma função *hash* - *h(k)*.

Assim, pode-se determinar o índice do elemento antes de realizar sua busca. Para o termo Ana, saberíamos de antemão, em qual elemento está, e não seria necessário percorrer toda a estrutura de dados para localizá-lo. A determinação do índice de acordo com os dados ocorre no momento de popular a tabela, conforme figura a seguir. Por exemplo:

Dado: Mia

Passo 1: Somar os valores do código Ascii de cada caractere

M = 77

i = 105

a = 97

Soma = 279

Passo 2: Obter o resto da divisão da soma pelo número de elementos: 279/11 = 25; resto = 4

Passo 3: Armazenar Mia no elemento 4

O mesmo ocorreria com Tim: 298/11 = 27; resto = 1

Figura 7 - O dado é inserido da tabela hash de acordo com um cálculo.
Fonte: Elaborada pelo autor, 2019.

VOCÊ O CONHECE?

Donald Knuth, nascido 10 de janeiro de 1938, nos Estados Unidos, é cientista da computação e professor da Universidade de Stanford. É considerado um dos criadores da análise de algoritmos, precursor da programação literária; desenvolveu o conceito de número surreal, dentre outros trabalhos que muito contribuíram para o desenvolvimento da computação. Para saber mais, leia o livro "The Art of Computer Programming". (KNUTH, 1973).

Para a busca de um valor, realiza-se o mesmo cálculo, o que determinará o elemento onde tal dado está. Bastando, então, recuperar o dado. Por exemplo, localizar Ada na tabela *hash* T:

Dado: Ada = 262 Mod 11 = 9

meuDado = T[9] //O elemento 9 contem Ada.

(Algoritmo 6 - fragmento: buscar por “Ada" na tabela hash T)

Colisões

De acordo com Googdrich e Tamassia (2013), Pode acontecer de diferentes chaves gerarem o mesmo índice (Ada mod 11 = 9; Acb mod 11 = 9). Esse fenômeno é chamado colisão e as tabelas hash os tratam de duas maneiras: Arranjo em buckets ou Endereçamento aberto.

Arranjo em buckets

O Arranjo em buckets é uma estrutura na qual cada célula da tabela *hash* é um contêiner para vários pares chave-valor. Assim, quando uma chave for entrada e o elemento do respectivo índice estiver ocupado, o contêiner receberá esse novo par (k.v), adicinando-o a uma lista de pares, ilustrada pela figura a seguir.

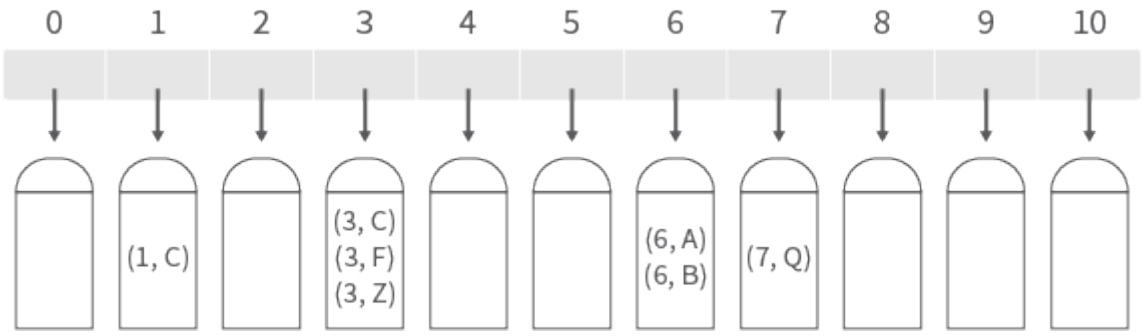

Figura 8 - Arranjo de buckets e entradas que geraram colisões.
Fonte: Elaborado pelo autor, 2019.

Endereçamento aberto

No endereçamento aberto, ao encontrar um elemento que já esteja ocupado, a função *hash* avança para o próximo elemento, até localizar um espaço vago para a os dados. A estratégia mais simples para endereçamento aberto é o "teste linear”. Ao tentar inserir um item (*k,v*) em uma posição A[*i*] que está ocupada, onde *i* = *h*(*k*), tenta-se de novo em A[*i*+1 mod *N*]. Se essa posição também estiver ocupada, tenta-se A[*i*+2 mod *N*] e assim sucessivamente, até que se encontre um espaço vago, como ilustrado na figura a seguir.

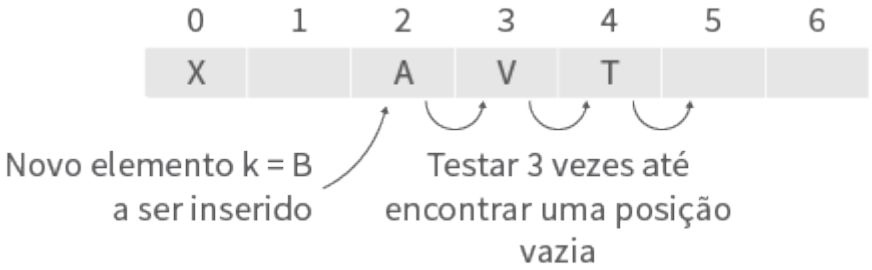

Figura 9 - Inserção em tabela hash com teste linear.
Fonte: Elaborada pelo autor, baseada em GOODRICH; TAMASSIA, 2013.

No caso do endereçamento aberto, a função de pesquisa (que localiza o item a partir de sua chave) também será ligeiramente diferente. Deve-se percorrer posições consecutivas, iniciando em A[$h(k)$] até encontrar o item com chave k ou um elemento vazio.

```
/*
 * Classe e método para contar palavras
 * com uso de Hashtable
 */
import  java.util.HashMap;
public class  contador_palavras {
     public  HashMap<String,Integer> contar(String texto) {
          HashMap<String, Integer> mapa = new  HashMap<String, Integer>();
          /*
           * A tabela hash tem uma String como chave e um inteiro como valor
           * Assim, cada palavra encontrada no texto terá uma entrada na tabela hash
           * E o seu valor representará a contagem de vezes que aparece no texto
          */
          for (String palavra : texto .split(" ")) {
               if(mapa.get(palavra)!=null){
                    //A palavra foi encontrada na tabela hash
                    mapa.put(palavra, mapa.get(palavra)+1); //Adiciona 1 ao valor que acompanha palavra
               }
               else {
                    mapa.put(palavra, 1); //inserir a palavra pela primeira vez na tabela
               }
          }
          return mapa;
     }
}
```

```
/*
 * Para testar o método que usa hash table
 * Contar palavras
 */
import  java.util.Map.Entry;

public class  teste_contar_palavras {

     public static void main(String[] args)  {
          contador_palavras cp = new  contador_palavras();
          String texto = "a tabela hash armazena a palavra na forma de entrada. A fução hash determina o
índice i";
          for( Entry<String, Integer> entrada : cp.contar(texto).entrySet()) {
               System.out.println(entrada.getKey() +"="+ entrada.getValue());
          }
     }
}
```

Figura 10 - Exemplo de implementação Java para tabela hash

Armazenamento externo

O volume de dados disponíveis cresce rapidamente e é um desafio permanente para o desenvolvimento de plataformas de *software*. A variedade de aplicativos cada vez mais intensivos em dados, incluindo bancos de dados, simuladores, cálculos científicos, computação gráfica, multimídia, sensores, mensagens instantâneas e correio eletrônico são exemplos de situações em que o volume de dados sempre é alto. Por exemplo, o *Google Earth* têm vários *Terabytes* (1000 *Gigabytes*) de tamanho. O data *warehouse* de vendas da *Wal-Mart* contém mais de meio *Petabyte* (500 *Terabytes*) de dados. O desafio aqui colocado está em desenvolver algoritmos para processar esses dados, ou então grande parte deles será inútil. Os computadores de uso geral organizam a

memória em diferentes níveis, passando de memórias internas, de nível mais baixo, até a memória externa, de nível mais alto, ilustrado pela figura a seguir.

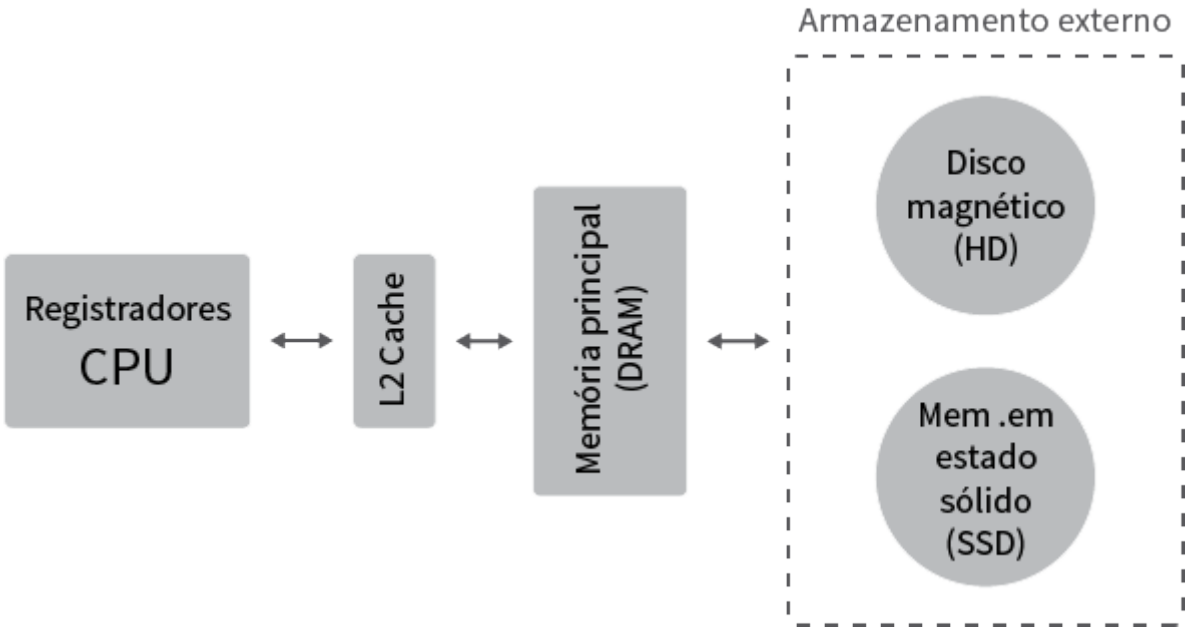

Figura 11 - Níveis da memória em um computador convencional.
Fonte: Elaborada pelo autor, 2019.

A CPU "*Central Processing Unit*" (Unidade Central de Processamento) é considerada o coração do computador, é formada por: unidade de controle; unidade lógica e Aritmética e pelos registradores, ou seja, memórias de alta velocidade que funcionam basicamente junto ao processador. Um pouco mais distante do procesador temos a memória cache que é mais lenta que os registradores, porém muito mais rápida que a memoria RAM. A memoria cache é utilizada para armazenar os dados mais utilizados pelo procesador em relação à memoria RAM, como forma de ganho de desempenho. A próxima memória mais distante do processador é a memoria RAM, sendo mais lenta que a memoria cache, serve para armazenar os dados durante a execução dos programas. Em outras palavras, todas estas memórias são utilizadas pelo processador, porém, em grau de importância temos: primeiramente os registradores, as memórias cache e a memoria RAM. É importante sabermos que todas estas memórias são consideradas internas, primárias e voláteis, ou seja, conservam as informações apenas enquanto o sistema estiver energizado, e apresentam baixa capacidade de armazenamento. Diferente a elas, temos as memórias externas ou memórias secundárias, também chamadas de memórias de massa; capazes de armazenar grande quantidade de informações, possuindo baixa velocidade de acesso, temos como exemplo destas: os discos magnéticos (HDs) e memórias em estado sólido (SSD).

CASO

No mundo moderno o termo algoritmo vem se tornando natural para a maioria das pessoas, sendo de extrema importância para que os computadores realizem de forma adequada suas tarefas, mas qual o melhor algoritmo? Como classificá-los? Os algoritmos apresentam grau de importância relativa, estando associados ao problema para o qual são criados, por exemplo, os algoritmos de busca sequencial são utilizados para sequência de dados desordenados. Imagine uma busca sequencial exaustiva em uma série desordenada de milhões de elementos, podendo haver necessidade de testar todos eles até encontrar o valor desejado. Mas se a mesma

sequência estiver ordenada, então poderiamos aplicar o algoritmo de busca binária, muito mais rápido, dividindo a série em duas e desprezando uma das partes, testando apenas aquela onde estivesse o valor desejado. A aplicabilidade deve ser analisada conforme o problema apresentado, ou seja, um determinado algoritmo pode ser excelente em dada situação e péssimo em outra.

A maioria das linguagens de programação considera modelos de programação nos quais a memória consiste em um espaço de endereço uniforme, cujo tempo de acesso é igual para todas as situações. Embora essa suposição seja razoável quando o conjunto de dados não é grande, ao processar grandes volumes de dados é importante lembrar que nem todas as memórias são construídas da mesma forma e, portanto, apresentarão diferentes comportamentos. O tempo de acesso aos dados chama-se latência e impacta de diferentes maneiras, conforme a memória vai sendo utilizada; acessar dados nos níveis mais baixos da memória é mais rápido. Por exemplo, carregar dados da memória interna (DRAM) pode levar alguns nanosegundos, mas a latência de acessar dados em um disco é de vários milissegundos, o que é cerca de um milhão de vezes mais lento. Nas soluções que processam ou realizam Entrada/Saída em grandes volumes de dados, a latência se torna um gargalo e pode, inclusive, impedir o funcionamento em determinadas situações. (VITTER, 2008).

Assim, os algoritmos para processamento e, especialmente, pesquisa em arquivos externos devem ser construídos de maneira a lidar com as limitações impostas pela memória interna, que não tem espaço para todos os dados armazenados externamente. Vitter (2008), coloca que os problemas de processamento e pesquisa em grandes volumes de dados se concentram em duas categorias:

Problemas em lotes, nos quais nenhum pré-processamento é realizado e todo o arquivo de itens de dados deve ser processado, geralmente com o intuito de transmitir os dados passando pela memória interna em uma ou mais etapas.

Problemas online, nos quais o processamento é feito em resposta para uma série contínua de operações de consulta (respostas são recebidas continuamente). Uma técnica comum para problemas on-line é organizar os itens de dados por meio de índice um hierárquico, de modo que apenas uma parcela pequena dos dados seja examinada em resposta a cada consulta. Os arquivos consultados podem ser estáticos, o que permite pré-processá-los para respostas eficientes; ou dinâmicos, nos quais as consultas são realizadas simultaneamente a atualizações, como inserções e exclusões.

Tipos de acessos

As memórias internas, conforme estudamos, são muito importantes para o funcionamento dos computadores. Atualmente, existe uma grande variedade de memórias internas, apresentando diversas características, dentre as quais, o tipo de acesso. Vejamos a seguir alguns tipos de acessos.

Acesso sequencial	O computador lê e grava segmentos contíguos de dados. Os dados são dispostos em espaços continuos e, para realizar uma pesquisa, o computador deve percorrer toda a estrutura, até que o dado seja encontrado. Então, o dado é carregado para a memória principal e um passo, caso seu tamanho seja pequeno suficiente para caber na memória, ou vários passos, caso os dados sejam grandes e não caibam na memória.
Acesso aleatório	O computador precisa realizar pesquisas para descobrir onde os dados estão. A busca pode ser em qualquer posição da memória. As memórias que possuem este tipo de acesso podem ser chamadas de memórias de leitura e escrita.

É importante considerar que há diferenças entre leitura e gravação, Por exemplo, dados que sejam gravados próximos não serão, necessariamente, lidos em sequência.

Árvores binárias (*B-Trees*)

As árvores binárias combinam o acesso aleatório com o acesso sequencial. Mantendo as chaves organizadas, a árvore binária permite e acesso aleatório durante a descoberta de onde os dados estão. Então, uma leitura sequencial é realizada, até que os dados terminem ou que o intervalo para leitura se esgote. Lembre-se: Chaves são as entradas de dados utilizadas para identificar um determinado item dentro da estrutura de dados.

São chamadas árvores binárias porque cada nó da árvore pode se dividir em somente dois caminhos. Dessa forma, para encontrar uma chave basta:

1) percorrer os nós perguntando se a chave foi encontrada;
2) caso a chave não seja encontrada : se a chave for menor que o valor do nó atual, tomar o caminho da esquerda e repetir o passo 1; se a chave for maior que o valor do nó atual, tomar o caminho da direita e repetir o passo 1.

Em todos os casos, o uso da memória externa leva a uma restrição no tocante a ordenação e pesquisa. Uma ordenação aleatória significa que a gravação será mais rápida, pois não é necessário determinar onde o dado deve ser gravado. No entanto, a pesquisa (para leitura) será prejudicada, porque a busca exigirá mais processamento. Uma ordenação sequencial dos dados gravados permite pesquisas mais rápidas, mas, ao gravar os dados, será necessário determinar o local onde os dados serão gravados e, eventualmente, movimentar o conteúdo de forma a criar o espaço correto para o dado. Por exemplo:

- Dados gravados = A-C-D
- Novo dados = B
- Necessário criar o espaço para B = A - - C - D
- Então gravar o B = A - B - C - D

O uso dos métodos adequados tanto para armazenamento quanto para ordenação e pesquisa determinam a eficiência do sistema desenvolvido, economizando recursos computacionais e, consequentemente, o custo de processamento dessas soluções.

1.1.2 Pesquisa

Pesquisar dados é uma tarefas mais comuns em desenvolvimento de *software*. Aplicações para mensagens instantâneas podem procurar por contatos ou por conteúdos. Sistemas de localização buscam por caminhos mais curtos. Um sistema empresarial busca dados no banco de dados. Os algoritmos de pesquisa fazem parte do nosso dia-a-dia e, muitas vezes, não percebemos. O desafio na pesquisa de dados é desenvolver a aplicação de busca da maneira mais eficiente possível, de forma que os processos subsequentes não sejam impactados por uma eventual demora. Essa eficiência deve ser a preocupação permanente do desenvolvedor. A evolução da ciência da computação é marcada pelo aprimoramento dos algoritmos de pesquisa, especialmente quando se considera que o volume de dados cresceu consideravelmente e que a complexidade dos processos é maior e que as pessoas têm menos tempo a perder. De forma geral, há duas estratégias de pesquisa: pesquisa sequencial e pesquisa binária.

Pesquisa linear ou sequencial

A pesquisa linear ou sequencial percorre todos os elementos de um vetor, até encontrar a chave. Então, a posição do item é retornada. Caso não encontre a chave, deve retornar um valor informando que não teve sucesso na busca, conforme figura a seguir.

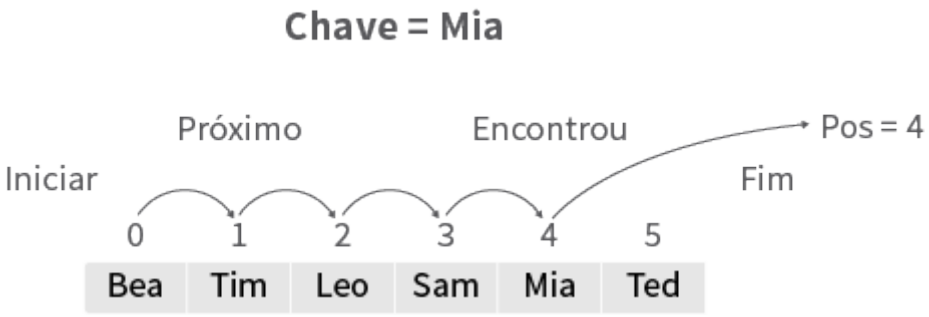

Figura 12 - Lógica geral da busca sequencial.
Fonte: Elaborada pelo autor, 2019.

Essa busca não tem grande eficiência, uma vez que, se o vetor tiver n elementos, é possível que a busca requeira n iterações para encontrar a chave desejada. No entanto, esta é uma alternativa para os momentos em que os dados não estejam ordenados ou nenhum outro algoritmo seja viável.
O pseudo código para a busca linear ou sequencial é:

```
Início
Int chave, retorno
Retorno = -1 //Se a chave não for encontrada, retornar -1
Int vetor[5] = {11,21,31,41,51}
Escreva "Informe um valor para busca"
Leia chave
Para i de 0 até 5, passo 1, faça
Se vetor[i]=chave entao
retorno = i
Fim_para //Parar o laço de repetição ao encontrar a primeira ocorrência
Fim_se
Fim_para
Escreva (Chave + " encontrada na posição " + retorno
Fim
```

(Algoritmo 5 - Pesquisa sequencial em um vetor unidimensional)

Pesquisa binária

A pesquisa binária começa examinando o item do meio da lista ordenada e caso não seja o valor procurado, podemos eliminar uma das metades. Trata-se de um algoritmo mais eficiente que a pesquisa linear, por reduzir o espaço de busca consecutivamente, restringindo as localizações possíveis.
O uso mais comum da pesquisa binária é buscar por uma chave em um vetor de dados. Suponha um vetor *v*, com um milhão de entradas numéricas. Dada uma chave *n*, uma pesquisa linear pode chegar, no pior caso, a um milhão de operações. Se, no entanto, considerarmos a lista que deve estar ordenada, dividindo-a ao meio, o valor procurado deve ser o elemento mediano ou estar em um dos dois segmentos obtidos, reduzindo o espaço a ser percorrido pela metade. Uma vez que cada novo segmento será dividido em dois, a redução se amplia, até que se encontre o valor desejado.
De forma geral:
divide-se o vetor em duas partes e compara-se o elemento mediano do vetor com o valor procurado;
se o elemento mediano for igual à chave: retorna-se a posição do elemento mediano;
termina-se o programa;
se o elemento médio for menor que o valor procurado: divide-se a parte à direita do elemento mediano, e compara-se com o valor procurado;
se o elemento médio for maior que o valor procurado: divide-se a parte à esquerda do elemento mediano, e compara-se com o valor procurado;
repete até esgotar o número de elementos ou até localizar o valor procurado.

As divisões consecutivas reduzem o espaço de pesquisa como ilustrado na figura abaixo:

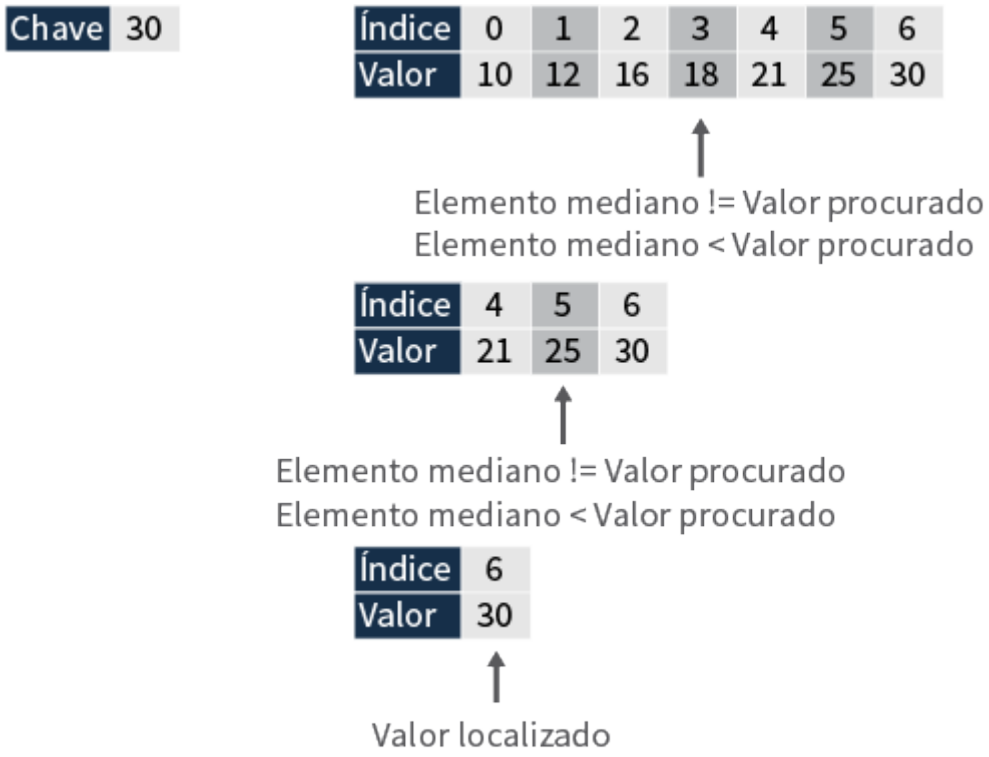

Figura 13 - Exemplo de uma pesquisa binária.

Para reduzir o vetor a ser pesquisado, basta manter três variáveis: uma com o índice do meio do vetor, uma com o índice do primeiro elemento do vetor e outra com o índice do último elemento do vetor.
O algoritmo para a pesquisa binária é escrito assim:

```
Início
inteiro v[5] = {10,12,16,18,21,25,30}, chave, tamanho
inteiro inicio, maior, final, resultado
resultado = -1 // -1 será o valor resultante quando a chave não for localizada
chave = 4 //Pesquisar por 4 no vetor v
tamanho = 7
início = 0
final = tamanho-1 // o índice do fim do vetor é sempre tamanho -1
enquanto inicio <= final faz
meio = (inicio + final)/2
se v[meio] = chave então
resultado =meio
fim_enquanto // Finalizar o laço para terminar a busca
fim_se
se v[m] < chave então
inicio = meio + 1 //Desprezando o segmento à esquerda
senão
Final = meio - 1 //Desprezando o segmento à direita
fim_se
fim_enquanto
escrever resultado
```

fim
(Algoritmo 6 - exemplo de busca binária)
É importante destacar que a busca binária requer, impreterivelmente, que os dados estejam ordenados. De outra maneira, não funcionará.

1.1.3 Ordenação

Ordenar dados é a operação de alterar um vetor de forma que ele obedeça a uma determinada ordem. (AZEREDO, 1996). A ordenação é uma das tarefas mais críticas relacionadas à pesquisa, uma vez que vários métodos de pesquisa não funcionam sem que os dados estejam previamente ordenados.
Além disso, a ordenação acrescenta processamento à aplicação, requerendo que o desenvolvedor possua domínio sobre diferentes métodos, para que a solução seja adequadamente eficiente. Há vários métodos para ordenação, cada um deles voltado para diferentes situações e com diferentes desempenhos, os mais frequentes são o *Bubble Sort* e o *Quick Sort*.

Bubble Sort

O algoritmo *Bubble Sort* ou ordenação por flutuação, também é conhecido como método da bolha. O método consiste em percorrer diversas vezes o vetor e a cada passagem fazer flutuar para o topo o maior elemento da sequência, daí o nome "método da bolha". Trata-se de comparar um dos itens do vetor com todos os demais. Quando o primeiro elemento é maior que o segundo, efetua-se a troca de posições. Essa comparação é realizada repetidamente, com cada um dos itens dos dados.

```
Linha 1:    int aux;
Linha 2:    int[] v = {4,1,3,2};
Linha 3:    for(int i=0; i<v.length; i++) {
Linha 4:        for(int j=0; j<v.length; j++) {
Linha 5:            if(v[i]<v[j]) {
Linha 6:                aux = v[i];
Linha 7:                v[i]=v[j];
Linha 8:                v[j]=aux;
Linha 9:            }
Linha 10:       }
Linha 11:   }
```

Figura 14 - Bubble sort, escrito em Java

Linha 3: Para cada elemento em *v*, percorre-se *i*.
Linha 4: Para cada elemento em v, percorre-se *j*.
Linha 5: Se o valor de v[*i*] for menor que o valor de v[*j*]
Linhas 6,7, 8: Substitui-se v[*j*] por v[*i*]
Ao executar o algoritmo, temos:
1ª iteração
v = {4, 1, 3, 2} (*i* = 0 *j* = 0)
2ª iteração
v = {4, 1, 3, 2} (*i* = 0 *j* = 1)
3ª iteração
v = {1, 4, 3, 2} (*i* = 0 *j* = 2)
4ª iteração

v = {1, 4, 3, 2} (i = 0 *j* = 3)
5ª iteração
v = {1, 4, 3, 2} (*i* = 1 *j* = 1)
6ª iteração
v = {1, 4, 3, 2} (*i* = 1 *j* = 2)
7ª iteração
v = {1, 3, 4, 2} (*i* = 1 *j* = 3)
8ª iteração
v = {1, 2, 4, 3} (*i* = 2 *j* = 2)
9ª iteração
v = {1, 2, 4, 3} (*i* = 2 *j* = 3)
10ª iteração
v = {1, 2, 3, 4} (*i* = 3 *j* = 3)

VAMOS PRATICAR?

Construa uma aplicação que ordene o vetor n mediante *Bubble Sort*.

Valores para n (copie estes valores):

{9, 20, 17, 10, 18, 25, 25, 15, 2, 15, 17, 17, 16, 11, 3, 11, 25, 16, 12, 22, 24, 14, 8, 16, 21, 27, 27, 18, 1, 29, 16, 10, 0, 2, 2, 26, 19, 9, 12, 24, 20, 3, 16, 4, 4, 11, 9, 21, 25, 6, 25, 10, 29, 20, 17, 23, 3, 26, 0, 30, 4, 20, 7, 11, 11, 19, 21, 4, 24, 13, 9, 29, 10, 22, 6, 28, 29, 28, 22, 10, 17, 3, 1, 1, 18, 2, 3, 11, 12, 28, 28, 7, 30, 25, 17, 28, 21, 12, 5, 12, }

Quick Sort

O *Quick Sort* é um algoritmo que adota a estratégia de divisão e conquista, ou seja, o método consiste em dividir uma sequência de valores em chaves, onde as chaves menores devem preceder as maiores. Posteriormente, as sublistas devem ser ordenadas recursivamente até que todos os valores estejam ordenados. Observem os seguintes passos:

Primeiro passo	Escolher um elemento da lista chamado pivô;
Segundo passo	Todos os elementos menores que o pivô devem ser rearranjados de modo a precede-lo e os elementos maiores devem ser posicionados em seguida;
Terceiro passo	Os elementos anteriores e posteriores devem ser recursivamnte ordenados.

Trata-se de um um algorítmo muito eficiente que reduz consideravelmente o tempo de ordenação, podendo ser utilizado em grandes quantidades de dados.

1.1.4 Análise de complexidade

O tempo de execução de um algoritmo depende do quão demorado é para um computador executar as linhas de código do algoritmo e isto depende da velocidade deste computador, da linguagem de programação, e do compilador que traduziu o programa da linguagem de programação para o código que executa diretamente no computador, entre outros fatores. Entretanto, se desconsiderarmos a capacidade de processamento do computador, ainda teremos a quantidade de iterações e de transposições que o algoritmo faz. (MANZANO; LOURENÇO; MATOS, 2015). Ainda de acordo com os autores, essa avaliação de quanto o algoritmo requer do computador para rodar, indiferentemente da capacidade do dispositivo, é chamada análise de complexidade. O primeiro aspecto a ser considerado é o tempo que o algoritmo leva para rodar de acordo com a quantidade de entradas no vetor de dados.

Para uma pesquisa linear, o melhor caso será quando a chave estiver na primeira posição e o pior caso será quando a chave não for localizada, pois o algoritmo percorrerá todos os elementos do vetor. Logo, a função de crescimento da busca linear é *n*. Ou seja, o tempo de execução é diretamente proporcional ao número de elementos. Para uma pesquisa binária, o melhor caso será quando a chave estiver no elemento mediano do vetor e, o pior caso, será quando a chave não for localizada. Mas, por conta da lógica da busca binária, dividindo o espaço de busca pela metade a cada iteração, o número de repetições é reduzido como apresentado no quadro abaixo. Para o caso da busca binária, a função de crescimento é *logn (logaritmo na base 2 de n)*, que é o resultado matemático das divisões consecutivas.

Numero de elementos	Repetições
2	1
8	3
20	4
200	8
2.000	11
2.000.000	21
2.000.000.000	31

Quadro 1 - Iterações necessárias (piores casos) para executar uma busca binária.
Fonte: Elaborado pelo autor, 2019.

Essa diferença entre as execuções da busca linear e da busca binária influencia diretamente no tempo de execução do algoritmo. A análise de complexidade busca comparar o desempenho de diferentes algoritmos com a finalidade de determinar qual lógica exige menos recursos. Considerando-se que o tempo de execução T(n), onde T é o tempo e n é o número de elementos processados, necessita-se de um modo para comparar a velocidade de crescimento de diferentes funções (que são a execução dos processos dos algoritmos de busca ou de ordenação). Essa comparação é denominada comparação assintótica.

É importante destacar que o tempo de execução também é influenciado por outros processos, por exemplo, a quantidade de decisões (if) e os processos de troca de dados são somados ao tempo. Assim uma equação de tempo de execução poderia ser:

$T(n) = n * 10$ ou $T(n) = n^2 + 3n - 2$

Entretanto, convenciona-se que a parte relevante da função é o n, pois a análise de complexidade se dedica a grandes volumes de elementos, portanto, a quantidade é importante. Assim, a função de crescimento é reduzida:
T(n) = n *10 equivale a f(n) = n

T(n) = n^2 + 3n -2 equivale a f(n) = n^2

Funções de crescimento	Total de Iterações
$f(n) = n^5$	1.000.000.000,00
$f(n) = n^3 + 4n - 2$	999.998.00
$f(n) = n + 10000$	2000
$f(n) = \sqrt{(n)}$	31
$f(n) = Log(n)$	7

Quadro 2 - Diferentes funções de crescimento e seus respectivos resultados.
Fonte: Elaborado pelo autor, 2019.

Observando o quadro acima, com diferentes funções, percebe-se que algumas delas tem maiores velocidades de crescimento. Para representar a complexidade, estabelece-se um limite que represente como a função de crescimento se comporta. A notação O determina a ordem da função e é uma das alternativas mais frequentes para essa representação. Escreve-se O(f(n)). A notação O é dada por:
Sejam T(n) e f(n) funções dos inteiros no conjunto real; afirma-se que T(n) é O(f(n)) se existem constantes positivas c e n_0 tais que: T(n) ≤ cf(n) para todo n ≥ n_0. Assim, se considerarmos um algoritmo em que T(n) = n^2 + 1000 e f(n) = n^2, podemos criar um gráfico para o intervalo até 50 elementos, ilustrado na figura abaixo.

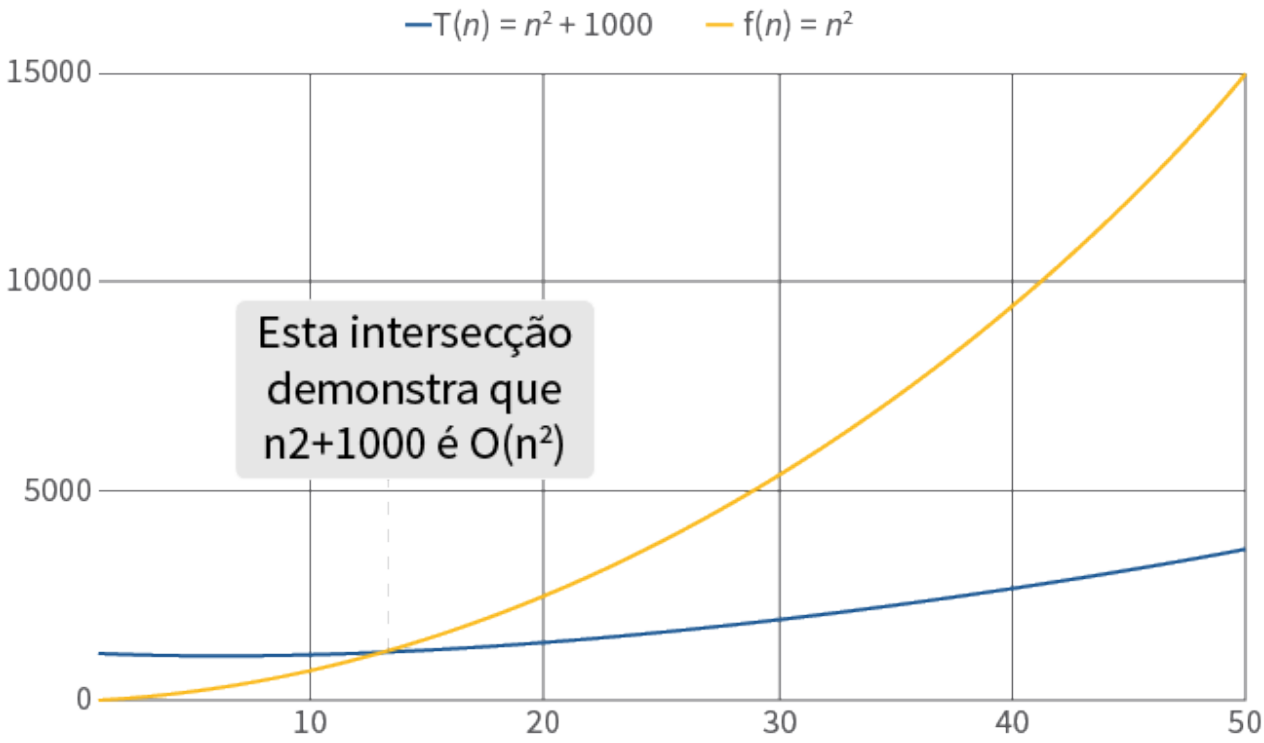

Figura 15 - Gráfico demonstrando que f(n) é O(n) - Ordem de n.
Fonte: Elaborada pelo autor, 2019.

Sempre que um algoritmo de ordenação ou de busca for implementado, deve-se considerar a sua eficiência mediante a avaliação da complexidade, com a finalidade de determinar se há alternativas mais eficientes para a solução.

1.2 Procedimentos de funções e funções recursivas

Os problemas relacionados a ordenação e pesquisa são complexos e sua implementação requer, em muitos casos, a construção de inúmeras linhas de código. No entanto, o aumento das linhas de código traz em si dificuldades de manutenção do programa; quando for necessário realizar um ajuste ou correção, haverá mais trabalho quanto mais linhas de código forem implementadas. Também ocorre o surgimento de código redundante, blocos de procedimentos distribuídos no programa mas que tem a mesma finalidade. Organizar o código dos programas de forma que a manutenção seja eficiente e não haja códigos redundantes é uma boa prática indispensável, atingida pela modularização e pela recursividade.

1.2.1 Modularização e reaproveitamento de código e funções

A modularização do código consiste em separar a lógica de programação em elementos distintos e com responsabilidades específicas. Por exemplo, um programa que realize operações matemáticas pode ser organizado de forma a simplificar seu uso mediante a construção de funções reaproveitáveis. (SOUZA *et al*, 2011).

```
public class modularizacao {

    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        System.out.println("Soma           = " +  somar(a,b));
        System.out.println("Subtração      = " +  somar(a,b));
        System.out.println("Multiplicação = " +  somar(a,b));
        System.out.println("Divisão        = " +  somar(a,b));
        // Se for necessário realizar uma operação, basta invocar a função desejada
        int c = 120;
        int d = 4;
        System.out.println("Multiplicação = " +  somar(c,d));
    }
    //As operações foram modularizadas em funções especializadas
    public static int somar(int a, int b) {
        return a+b;
    }
    public static int subtrair(int a, int b) {
        return a-b;
    }
    public static int multiplicar(int a, int b) {
        return a*b;
    }
    public static int dividir(int a, int b) {
        return a/b;
    }
}
```

Repare que cada módulo recebe dois argumentos (números inteiros), processa os dados e retorna o resultado. Esse retorno é capturado na instrução que invocou (chamou) a função.

VAMOS PRATICAR?

1. Crie uma função que receba duas strings e retorne sua concatenação => concat(v1, v2) retornará v1 + v2.
2. Agora, crie um programa que receba o nome e o sobrenome do usuário em duas variáveis diferentes e as concatene (mostre na tela ou no console).
3. Altere a ordem da concatenação, dentro da função (v2 + v1).
4. Rode o programa do passo dois novamente. Veja a diferença.

A cada vez que o programa principal precisa realizar uma operação matemática, basta chamar a função necessária. (DEITEL; DEITEL, 2015) . Por outro lado, se precisarmos alterar a lógica de alguma das funções, essa alteração de código se dará uma vez e será refletida em todos os procedimentos que utilizem tal função.

1.2.2 Recursividade

Muitos algoritmos executam o mesmo procedimento repetidas vezes, até que uma condição de parada seja alcançada. Por exemplo, o cálculo do fatorial de um número (*n*!):
4! = 4 x 3 x 2 x 1 = 24
Observado atentamente, o fatorial é:
4! = 4 x (4-1) x (3-1) x (2-1) = 24
Uma possível implementação do fatorial, utilizando funções é:

```
public class fatorial {

    public static void main(String[] args) {
        System.out.println(fatorial(4));
    }

    //Função para reaproveitar o cálculo do fatorial
    public static int fatorial(int n) {
        for(int i = n; i>1; i--) {
            n = n * (i-1);
        }
        return n;
    }
}
```

VOCÊ QUER LER?

A recursividade vem sendo aplicada largamente na programação de computadores para resolver problemas complexos. No artigo "A História da Ciência e o ensino da recursividade: As torres de Hanói", você encontrará um estudo sobre a natureza da recursividade, e algumas de suas aplicações. Leia o artigo completo neste link <https://revistas.pucsp.br/hcensino/search/search?simpleQuery=recursividade&searchField=query>.

Uma função recursiva é um procedimento que chama a si mesmo, resolvendo problemas sucessivamente, simplificando o problema até que se atinja uma condição de parada. A estrutura de uma função recursiva consiste no processamento de um caso base, que significa o final da pilha de execuções; e o processamento recursivo, que é chamada para ela mesma, com o problema simplificado.

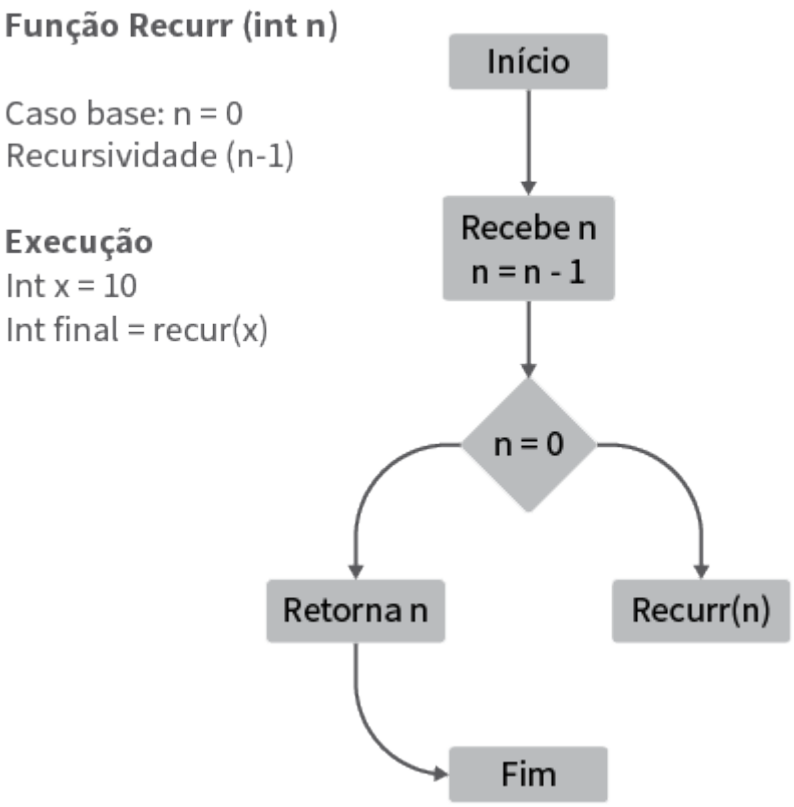

Figura 16 - Esquema lógico de uma função recursiva.
Fonte: Elaborada pelo autor, 2019.

A função recurr(), ilustrada na figura acima, é um exemplo simples, que tem por finalidade subtrair um valor até que ele chegue em 0 (zero). Ao chegar em 0 (zero), a função para e retorna n - este é o caso base. Enquanto o valor de n for maior que 1, a função recurr é chamada novamente.
O código para a função recurr() é:

```
public class recursividadae {
public static void main(String[] args) {
int i =10;
System.out.println(recurr(i));
}
public static int recurr(int n) {
n = n-1;
if(n>0) {
return recurr(n);
}
return n;
}
}
```

Então a mesma técnica pode ser aplicada para resolução do fatorial de n (n!), uma vez que a cada chamada subtrai-se 1 e o caso base é n= 1. (ZIVIANI, 2012).

```
public class fatorial_recursivo {
public static void main(String[] args) {
int numero = 8;
int fat = fatorial(numero);
System.out.println(fat);
}
```

```
public static int fatorial(int n) {
if (n==0) {
return 1; //0! é igual a 1
}
return n*fatorial(n-1);
}
}
```

VAMOS PRATICAR?

1) Implemente a sua função recursiva para calculo fatorial

2) A seqüência de Fibonacci é um modelo famoso por representar várias estruturas naturais. Ela se dá pela equação f = (n-1) + (n-2): 1, 1, 2 ,3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Trata-se de um problema perfeito para solução recursiva.

Dado que o caso base é n < 2, implemente uma função recursiva para a sequência de Fibonacci.

O programa principal deve ter a seguinte estrutura:

```
for (int i = 0; i < 30; i++) {
System.out.print("(" + i + "):" + fibo(i) + " ");
}
```

A função recursiva é um procedimento símples, porém poderoso, capaz de minimizar o código, resolvendo questões complexas. O programador que souber utilizar adequadamente este recurso, elevará sua capacidade criativa e certamente se tornará um profissional mais valorizado.

Síntese

Concluímos a unidade abordando busca, ordenação e armazenamento de informações, sendo apresentada a importancia do tema no contexto atual, facilitando a construção de uma visão crítica sobre a realidade computacional.

Nesta unidade, você teve a oportunidade de:

- compreender o que é busca de dados, no que consiste e para que serve, observando exemplos práticos;
- conhecer métodos de ordenação, cujas características ficaram bem evidenciadas, facilitando sua visão crítica;
- entender as técnicas de armazenamento de dados, e os diferentes tipos de memórias;
- compreender complexidade de algorítmos.

Bibliografia

AZEREDO, P. A. **Métodos de Classificação de Dados e Análise de suas Complexidades**. Rio de Janeiro: Campus, 1996.

COSTA, E. B. L. A História da Ciencia e o Ensino da Recursividade: As torres de Hanoi. **História da Ciência e Ensino Construindo interfaces**. v. 4. p. 38 – 48. 2011. Disponível em: <https://revistas.pucsp.br/hcensino/article/view/5489/5768>. Acesso em: 01/10/2019.

DEITEL. P.; DEITEL H. **Java: Como Programar**. 8. ed. São Paulo: Pearson, 2015.

FEOFILOFF, P. **Algoritmos em linguagem C**. Rio de Janeiro: Elsevier, 2008.

GOODRICH, M. T.; TAMASSIA, R. **Estruturas de Dados e Algoritmos em Java**. 5. ed. Porto Alegre: Bookman, 2013.

GOVERNO DO ESTADO DE SÃO PAULO. Centro Paula Souza. Grupo de Estudos de Educação à Distância. Informática. Módulo I. Agenda 15. **Lógica de Programação: Vetores e Matrizes**. 13:09 min. Canal GEEaD CPS, 2014. Disponível em: <https://www.youtube.com/watch?v=7i3YfiLy1vE&list=LLecA4jNMub1jeYn7rRe__gA&index=2&t=0s>. Acesso em: 01/10/2019.

KNUTH, D. E. **The Art of Computer Programming**. v. 1. 2. ed. Massachusetts: Addison-Wesley, 1973.

MORAIS, I. S. *et al*. **Algoritmo e programação (engenharia)**. Porto Alegre: SAGAH, 2018.

MANZANO, J. A. N. G.; LOURENÇO, A. E.; MATOS, E. **Algoritmos - Técnicas de Programação**. 2. ed. São Paulo: Érica, 2015.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. **Algoritmos: lógica para desenvolvimento de programação de computadores**. 28. ed. São Paulo: Érica, 2016.

RIBEIRO, J. A. Introdução à programação e aos algoritmos. 1. ed. Rio de Janeiro : LTC, 2019.

SOUZA, M. A. F.; GOMES, M. M.; SOARES, M. V.; CONCILIO R. **Algoritmos e lógica de programação. Um texto introdutório para a engenharia**. 3. ed. São Paulo: Cengage Learning, 2011.

VITTER, J. S. **Algorithms and Data Structures for External Memory**. Boston: Now Plubishers, 2008.

ZIVIANI, N. **Projeto de Algoritmos: com implementações em JAVA e C++**. São Paulo: Cengage Learning, 2012.