Armazenamento e Recuperação da Informação

UAB-UFSCar
Universidade Federal de São Carlos
Rodovia Washington Luís, km 235
13565-905 - São Carlos, SP, Brasil
Telefax (16) 3351-8420
www.uab.ufscar.br
uab@ufscar.br

Jander Moreira

Armazenamento e Recuperação da Informação

São Carlos

2011

Concepção Pedagógica
Daniel Mill

Supervisão
Douglas Henrique Perez Pino

Equipe de Revisão Linguística
Ana Luiza Menezes Baldin
Jorge Ialanji Filholini
Paula Sayuri Yanagiwara
Priscilla Del Fiori
Sara Naime Vidal Vital

Equipe de Editoração Eletrônica
Christhiano Henrique Menezes de Ávila Peres
Izis Cavalcanti
Rodrigo Rosalis da Silva

Equipe de Ilustração
Jorge Luís Alves de Oliveira
Priscila Martins de Alexandre
Thaisa Assami Guimarães Makino

Capa e Projeto Gráfico
Luís Gustavo Sousa Sguissardi

SUMÁRIO

APRESENTAÇÃO

A visão do desenvolvimento de um programa para resolver problemas passa pelos conceitos de algoritmos e do uso adequado de uma linguagem de programação. Mas isso não basta; é preciso conhecer as diversas estruturações e técnicas para a manipulação de dados.

Este livro cobre diversos aspectos essenciais para uma manipulação adequada dos dados.

O primeiro passo, neste universo, passa pelas noções de análise de algoritmos. Para entender algoritmos, não basta medir o tempo que ele leva para executar, pois é preciso conhecer seu comportamento como um todo e saber como a variação da quantidade de dados que será processada influenciará no custo de tempo desta execução. A análise de algoritmos, neste contexto, inclui entender como cada instrução de um algoritmo ou programa influencia no tempo de execução. Laços de repetição executados diversas vezes aumentam o tempo de execução, enquanto comandos condicionais podem determinar se partes custosas dos algoritmos são ou não executadas. Dependendo dos dados, os algoritmos também podem ter desempenhos diferentes. Por exemplo, ordenar uma coleção de dados que esteja ordenada em partes tende a ser mais simples do que encarar uma coleção completamente embaralhada.

Ordenar conjuntos de dados é importante nas organizações de dados. Localizar itens em coleções ordenadas é sempre mais eficiente do que em coleções sem ordenação. Desta forma, técnicas clássicas de ordenação em memória principal são abordadas, cobrindo métodos mais triviais com desempenho menor e também métodos mais sofisticados, cujos desempenhos são significativamente melhores. Por outro lado, quando os dados são armazenados em memória secundária, o problema é diferente. Entender as implicações dos acessos feitos aos discos rígidos torna-se crítico para se ter eficiência na ordenação de grandes arquivos de dados.

O conhecimento da análise de algoritmos e a manipulação para ordenação dão base para as organizações de dados. Inicialmente é preciso compreender como os dados são representados e como efetivamente os armazenamentos em memória principal e em memória secundária se apresentam, tanto em termos de semelhanças quanto de diferenças.

Dados armazenados em memória principal, quando consideradas coleções relativamente grandes, são usualmente organizados em arranjos (vetores) de registros. Esses registros podem ser mantidos ou não de forma ordenada. Entender como diferentes opções de organização influenciam no desempenho é essencial para as escolhas. Como um conjunto de dados é organizado pela primeira vez?

Como são adicionados novos registros a este conjunto? Como e quais os custos das remoções? Como podem ser feitas as buscas por informações? Como cada organização se comporta com a continuidade do seu uso, considerando-se que a eficiência pode ser comprometida com o tempo? Cada uma dessas perguntas, se respondida adequadamente, permite ao desenvolvedor optar pela melhor organização para cada situação que se apresenta. Nem sempre as opções mais sofisticadas são as melhores para todos os casos. Entender quando aplicar soluções diferentes é o ponto chave para soluções mais ou menos adequadas.

Ao contrário da memória principal, para a qual o acesso aos dados é rápido, a memória secundária (em específico os discos rígidos) possui características estruturais diferentes e o custo de tempo para acessar os dados é significativamente maior. As formas de organização de dados para discos também será coberta neste livro. Se o custo de tempo de se chegar à informação é um ponto crítico no desempenho, então técnicas, estruturações e algoritmos que considerem a relevância deste fato devem ser considerados. Portanto, dados armazenados em disco também podem não ter nenhuma ordenação ou seguirem um critério de ordem crescente para um dado atributo. Os custos para se chegar aos dados para as diferentes estratégias de organização são também diferentes. Considerar como um novo registro é inserido ao conjunto e como uma remoção é realizada é, também, um ponto de interesse. Arquivos não ordenados e arquivos ordenados são cobertos sistematicamente, avaliando seus pontos positivos e negativos. Além destes, uma organização que envolve uma estrutura auxiliar para as buscas é considerada e, desse modo, uma visão sobre indexação é considerada.

Um último ponto, mas não menos importante, é abordado: a compressão dos dados. Manter os mesmos dados usando menos bytes é importante tanto para reduzir o espaço necessário para armazenamento quanto para reduzir a banda em uma transmissão pela rede. Embora a compressão de dados seja um tópico bastante amplo, podendo envolver perda de qualidade (como é o caso da compressão de músicas com o MP3), o foco deste livro são as noções de compressão sem perda, a qual pode ser aplicada a dados como catálogos, listas de funcionários e clientes, documentos, etc. Assim, a compressão é vista em como representar uma mesma informação usando menos bits.

De forma geral, o conteúdo deste livro complementa conceitos de algoritmos, programação e estruturas de dados, dando bases sólidas para entender aspectos importantes de bancos de dados e aplicações que exijam volumes de dados consideráveis.

UNIDADE 1

Análise de algoritmos

1.1 Primeiras palavras

A análise de algoritmos é um assunto extenso e será abordado neste livro em seus conceitos básicos. O foco é entender o comportamento dos algoritmos em função da carga de dados que deve ser processada.

Analisar um algoritmo permite avaliar seu custo, o que pode ser feito tanto em termos de tempo quanto de espaço. A *análise de tempo* se refere a avaliar como o tempo de execução de um algoritmo específico se comporta em função, por exemplo, da quantidade de dados que devem ser processados. Por sua vez, a *análise de espaço* dá suporte à avaliação da quantidade de memória que é necessária para que o algoritmo seja executado.

Em termos de tempo de execução, tome o exemplo de um algoritmo para determinar o maior elemento presente em um vetor de números reais. Supondo que tal algoritmo faça uma varredura, de elemento a elemento, comparando com o maior encontrado até o momento, não é complicado perceber que cada elemento é consultado apenas uma vez. Se o tempo de execução para um arranjo com n elementos leva $T(n)$ unidades de tempo[1], o tempo de execução para o dobro de elementos seria aproximadamente $2T(n)$, visto que apenas haveria mais elementos e o tempo para consultar cada um é praticamente o mesmo. Porém nem todos os algoritmos seguem esta lógica. Algoritmos que trabalham com ordenação de dados, por exemplo, exigem várias comparações dos elementos entre si e, ao dobrar o número de elementos, o tempo gasto passa a ser bem maior que o dobro do tempo anterior. Como ilustração, para algoritmos triviais de ordenação, se o número de elementos for duas vezes maior, o tempo consumido será praticamente multiplicado por quatro. Ou, ainda, se o número de elementos for 10 vezes maior, o tempo de execução será 100 vezes o tempo original.

No caso de análise de espaço, alguns algoritmos, para serem executados, exigem pouca memória adicional. Voltando ao exemplo do algoritmo usado para determinar o maior valor dentro de um vetor, bastam algumas variáveis para o laço de repetição e para manter o maior valor encontrado, além do próprio arranjo onde estão armazenados os dados. Se o número de elementos que devem ser verificados aumentar, não são necessárias mais variáveis ou mais espaço (exceto para o arranjo em si). Por outro lado, há algoritmos de ordenação que, dado um vetor com n elementos, sua ordenação exige a alocação de outro vetor de mesmo tamanho (para transferência dos dados de um para outro). Neste caso, para vetores muito grandes, a memória para executar o algoritmo pode

1 Uma *unidade de tempo*, aqui, é um tempo arbitrário qualquer. Em programas reais (e não algoritmos), equivaleria ao número de nanossegundos (milissegundos ou segundos) levados para a execução.

não ser suficiente, pois pelo menos dois vetores devem coexistir (o original e o auxiliar).

Embora ambas as formas de análise representem pontos importantes para a avaliação de algoritmos, aborda-se aqui a análise de tempo, cuja aplicação prática parece ser mais relevante que a que se refere ao espaço. Assim, embora o custo de espaço seja relevante, o enfoque é sobre a análise de tempo.

1.2 Comparação entre algoritmos e análise de complexidade

A análise de algoritmos em termos de tempo (referenciada a partir de agora somente por análise de algoritmos) permite determinar, de forma geral, o comportamento do algoritmo na medida em que a "carga" de dados que deve ser processada aumenta. Isso permite uma avaliação geral de o quanto um dado algoritmo é "bom" ou "ruim".

Thomas Fuller já disse, entretanto, que "nada é bom ou ruim, exceto por comparação". Para algoritmos, se seu tempo é considerado pesado (ou seja, ruim), mas este for o único algoritmo disponível para resolver o problema, então não se pode estabelecer um critério de comparação, de forma que termos como "melhor" ou "pior" deixam de fazer sentido. Avaliar o tempo de execução de algoritmos serve para permitir que, dados dois ou mais algoritmos que resolvam um mesmo problema, determinar qual tem comportamento melhor ou pior, ou seja, qual tende a consumir mais ou menos tempo, dependendo da carga de dados.

A realização de tais avaliações não depende apenas dos algoritmos em si, mas também da quantidade de dados que deve ser processada. Há exemplos de algoritmos que, apesar de serem julgados como ruins pela análise, podem ser melhores que os julgados bons, desde que a quantidade de dados seja pequena. Voltando ao exemplo de algoritmos que ordenam dados, se o conjunto de dados for pequeno (10 ou 20 elementos, por exemplo), praticamente qualquer método é muito rápido, mesmo os julgados pela análise como muito ruins. Neste caso, não importaria a análise.

Portanto, entender a situação é de suma importância para escolher a melhor alternativa para se resolver um dado problema, e não só se basear na análise realizada.

1.2.1 A complexidade dos algoritmos

A formalização da análise de tempo de um algoritmo passa por definir uma medida de tempo de execução. O nome *complexidade de algoritmos* é dado ao

processo de avaliação dos comandos do algoritmo de forma a estimar seu tempo de execução em função de sua carga.

A “carga de dados” que um algoritmo deve processar recebe o nome de *tamanho do problema*.

Uma forma clara de visualizar o tamanho do problema é o caso do processamento de dados armazenados em vetores na memória. Somar os dados de um vetor de números reais consome um tempo que depende claramente do número de elementos presentes no arranjo. Quanto mais itens a serem somados, maior o tempo de execução.

Porém, nem sempre é a quantidade numérica de dados que influencia no tempo de execução. Para se calcular o fatorial de um valor inteiro, por exemplo, o tempo não varia com a quantidade de dados (no caso, apenas um número), mas de acordo com a magnitude do valor de entrada: quanto maior o valor, maior o número de multiplicações até se chegar ao resultado desejado. Similarmente, o cálculo do maior divisor comum (MDC) de dois números inteiros positivos também depende da magnitude desses valores e não da quantidade (pois são sempre dois).

Um ponto importante a ser ressaltado, portanto, é que o primeiro passo para entender ou fazer uma análise é determinar o que representa, no algoritmo, o tamanho do problema. Outra ação importante é verificar outros fatores que consumam tempo, como é o caso do número de trocas que devem ser feitas, ou o número de multiplicações, ou ainda o número de comparações.

1.2.2 O que são o melhor caso, o pior caso e o caso médio?

A aplicação de algoritmos a determinados conjuntos de dados apresenta dependência do tempo não somente em relação à quantidade de instruções que devem ser executadas, mas também da situação dos dados.

Na situação do cálculo do fatorial de um número qualquer, não há qualquer dependência de outros fatores, a não ser do valor do número. Agora, assumindo que o problema seja colocar as cartas de um baralho em ordem, ou seja, para cada naipe, separar as cartas iniciando no *ás* e terminando no *rei*, de forma sequencial (começando com *espadas*, depois *copas*, seguido de *paus* e, finalmente, *ouros*) há algumas dependências a serem consideradas. Se as cartas já apresentarem certa ordenação, o trabalho será facilitado, pois menos movimentações serão necessárias. Em outras palavras, ordenar um baralho já com certa ordenação é, em geral, mais rápido que ordenar um que esteja bem embaralhado. É importante notar que uma mesma estratégia (algoritmo) para

ordenar as cartas deve ser usada, independentemente da situação do baralho, para que seja possível fazer esta comparação.

Então, pode-se dizer que há algoritmos que resolvem um determinado problema, mas que dependem apenas do tamanho do problema em questão. Outros, por outro lado, empregam estratégias que funcionam melhor dependendo de como os dados estão dispostos. Os primeiros não possuem “casos”, mas os do segundo tipo podem ter situações favoráveis, desfavoráveis ou ordinárias.

Quando os dados influenciam no comportamento do algoritmo, dá se o nome de *melhor caso* à situação em que os dados favorecem a execução do algoritmo, tornando-o mais rápido. O *pior caso* ocorre quando os dados são desfavoráveis ao algoritmo, tornando-o mais custoso computacionalmente, tornando-o mais lento. Em muitas ocasiões, é possível determinar uma situação não tendenciosa para os dados, situação em que o algoritmo se comporta de forma “normal”. Nesta última situação ocorre o *caso médio*.

Um exemplo pode ajudar no entendimento desses conceitos, supondo a existência de um algoritmo simples para localizar um dado número em um vetor de valores reais, no qual não há elementos repetidos.

O algoritmo de busca faz uma repetição iniciando as comparações a partir do primeiro elemento e indo um a um, até encontrar ou chegar ao último. Se houver uma tendência, por uma razão qualquer, de que os valores pesquisados se concentrem no início do vetor, então a busca é mais rápida, pois o número é localizado com poucas comparações. Algumas técnicas de pesquisa podem fazer com que itens pesquisados migrem em direção ao início do arranjo, de forma que os itens pesquisados com mais frequência tendam a ser localizados mais rapidamente. Assim, segundo estas circunstâncias, a pesquisa tem seu melhor caso quando os dados pesquisados se encontram no início do arranjo. É importante notar que, para a situação descrita, a probabilidade de acesso a determinados itens é maior do que a de outros; se todos os itens tiverem a mesma chance de serem pesquisados, então estas considerações perdem a validade.

Porém, se a maioria das pesquisas for por números que não estejam presentes no vetor, então a tendência é que todos os valores tenham que ser consultados até que se determine a ausência do item procurado. Esta é a situação de pior caso, na qual todas as possíveis comparações têm que ser feitas, gerando o maior custo computacional.

Se não houver nenhuma tendência nos dados, ou seja, considerando que qualquer valor presente no arranjo tenha a mesma probabilidade de ser buscado pelo algoritmo, pode-se considerar que, em média, metade dos itens tem seu valor comparado na pesquisa. Existem poucas comparações quando é achado mais para o início do vetor e mais quando é localizado mais para o final. Para

um grande número de pesquisas, a média de itens consultados equivale à metade do número de elementos do vetor. Este é o caso médio.

Há, ainda, situações em que existem outras tendências nos dados e, para elas, devem ser feitas considerações de cunho estatístico.

Independentemente da situação, cada caso é analisado separadamente. Assim, havendo dependência dos dados, os algoritmos podem ter análises diferentes para cada caso, sendo cada uma feita desconsiderando as demais, resultando em análises individuais. Em outras palavras, há uma análise para o melhor caso, outra para o pior caso e ainda uma terceira para o caso médio.

1.3 Complexidade assintótica

A análise dos algoritmos é feita, em termos de tempo, sobre o custo computacional que os comandos determinam.

Do ponto de vista deste livro, a análise fará considerações simplificadas sobre as demandas de tempo de cada comando. Ao final será visto que, para a análise, é importante o comportamento geral do algoritmo e não os muitos detalhes envolvidos, de modo que as simplificações não implicam em perda de qualidade na análise.

Para cada análise feita, será definido o que é o tamanho do problema para o algoritmo. Por convenção, essa característica será indicada pela variável fictícia *n*. É importante notar que *n* não é uma variável do algoritmo, mas a indicação do tamanho do problema. Assim, para algoritmos de pesquisa em vetores, *n* é o número de elementos presentes no vetor; no caso do cálculo do fatorial, *n* é o valor para o qual se calculará o fatorial; em pesquisa de dados em arquivos, *n* está relacionado ao número de registros armazenados; em pesquisas em árvores AVL, *n* se refere ao número de nós que a árvore possui; em outros problemas, *n* pode ser até a distância entre dois valores. Esses são apenas alguns exemplos, já que tudo depende do algoritmo em questão.

O tempo consumido por um algoritmo é função do tamanho do problema. A notação adotada para o tempo de execução é T(*n*), ou seja, o tempo em unidades de tempo arbitrárias em dependência de *n*.

A avaliação das características de T(*n*) é chamada *análise de complexidade* do algoritmo. Como, em geral, é mais importante saber como os algoritmos se comportam em relação a grandes volumes de dados (ou para tamanhos de problema grandes), é empregada a análise para valores grandes de *n*, o que caracteriza a chamada *complexidade assintótica*. Em outras palavras, são considerados os casos de T(*n*) em que *n* “tende a infinito”; na prática, quando *n* tem valores considerados altos.

Como exemplo, não são tão importantes as pesquisas em conjuntos de dados com 200, 300 ou 1000 itens, mas naqueles com milhares ou milhões de elementos. São relevantes as recuperações de registros em bases de dados com muitos (milhares) itens cadastrados. Seria o caso de ordenar dados de listas telefônicas e não de agendas pessoais, por exemplo.

Nos tópicos seguintes são abordados alguns conceitos matemáticos envolvidos na análise da complexidade e a avaliação do tempo de execução de algoritmos.

1.3.1 Aspectos formais e notação

Neste tópico são considerados os aspectos mais formais da complexidade de algoritmos, com introdução das notações principais.

O tamanho do problema, indicado por n, corresponde a um valor inteiro que reflete uma grandeza da qual o tempo de execução do algoritmo depende. Esse valor é considerado maior ou igual a zero. A função que reflete o tempo de execução em função de n é dada por T(n), considerada sempre positiva.

A complexidade utiliza uma notação específica, conhecida como “o-grande” ou “big-oh”, escrita na forma O($f(n)$). Esta notação pode ser lida como “O de f(n)” ou “big-oh de f(n)”. Por exemplo, um algoritmo de busca sequencial tem tempo de execução O(n) (“O de n”, ou ainda “da ordem de n”), enquanto métodos de ordenação podem ser O($n$log n)[2] ou O(n^2) (“da ordem $n$log n”; “big-oh de n^2”).

Essa notação informa uma característica global sobre o comportamento do tempo de execução de um algoritmo. Assim, um dado algoritmo ser avaliado como O(n^2) significa que, para valores grandes de n, o tempo de execução T(n) se comporta como uma parábola. Isso permite avaliar que, se o tamanho do problema for multiplicado por 10, o tempo de execução será aumentado em cerca de 100 vezes: T(10n) seria aproximadamente equivalente a 100 T(n).

Estas notações não são precisas, pois visam avaliar o comportamento do algoritmo e não medir exatamente quanto tempo gastaria em uma execução real. É mais importante, neste contexto, saber que um algoritmo O($n$log n), que faz a mesma coisa que outro O(n^2), tende a ter menor tempo de execução se houver aumento significativo do tamanho do problema.

É essencial que se tome cuidado com a notação, que é escrita simplesmente por O(f(n)), e não O(n) = f(n). Portanto, escrever que O(n) = n^2 não tem

2 A notação logarítmica usada neste texto sempre assume base 2. Assim, ao se escrever log n, assume-se por padrão $\log_2 n$. Quando necessário, outras bases ficarão explícitas.

significado nenhum porque O() não é uma função, mas uma notação específica. Assim, escreve-se apenas $O(n^2)$.

Matematicamente, uma função $T(n)$ pertence a $O(f(n))$ se, e somente se, $T(n) \leq cf(n)$ para todos os valores de $n \geq n_0$, com c e n_0 positivos e arbitrários.

A interpretação dessa definição pode ser feita da seguinte maneira: “se, a partir de um dado tamanho de problema, o tempo de execução do algoritmo $T(n)$ for limitado superiormente (ou seja, for menor ou igual) por uma função $cf(n)$, então a notação de qualificação $O(f(n))$ pode ser usada para indicar a complexidade do algoritmo”.

A aplicação deste conceito a comandos de algoritmos é apresentada na próxima seção. Para exemplificar a notação no sentido matemático, serão abordados aqui alguns exemplos.

Supondo que $T(n)$ seja dado por $T(n) = 50n + 20$, é possível mostrar que $T(n)$ pertence a $O(n)$. Para isso, basta mostrar que existem valores para c e n_0 que façam com que $T(n) \leq cn$, para valores $n \geq n_0$. Assim, a condição $50n + 20 \leq cn$ tem que ser válida para $n \geq n_0$. Se forem escolhidos c igual a 51 e n_0 igual a 20, então a condição se torna verdadeira, o que mostra que a notação pode ser usada corretamente. Isso significa que o tempo de execução $T(n)$ nunca se comporta pior que uma reta para valores grandes do tamanho do problema. A Figura 1 mostra o comportamento das duas funções e como $T(n)$ está abaixo de $51n$ à medida que n aumenta.

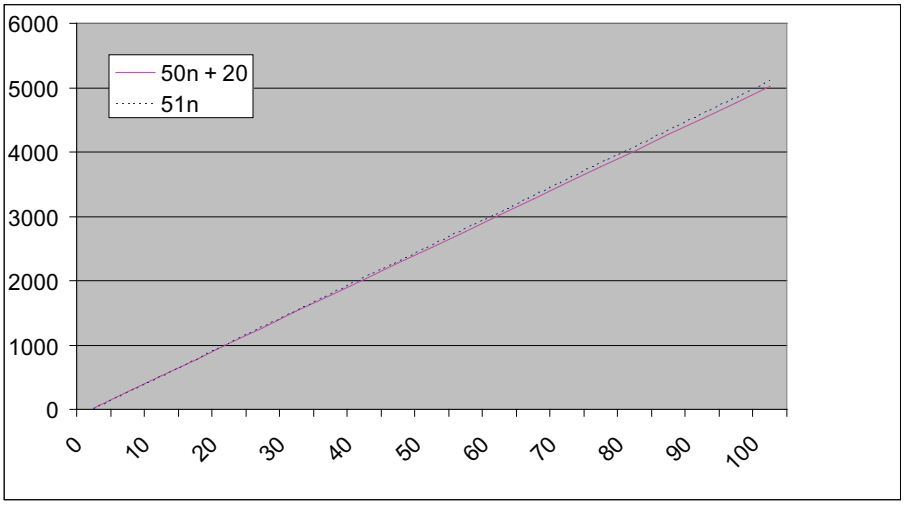


Figura 1 Gráfico mostrando que $51n$ (linha pontilhada) é um limitante superior para $50n + 20$ (linha sólida).

É possível também mostrar que $T(n) = 10n^2 + 3n + 4$ é $O(n^2)$. Basta achar valores c e n_0 que façam que $10n^2 + 3n + 4 \leq cn^2$ a partir de $n \geq n_0$. Para $c = 11$,

$T(n) \leq 11n^2$ para todo $n \geq 4$, o que comprova ser verdadeira a afirmação inicial. Essas duas funções são apresentadas na Figura 2.

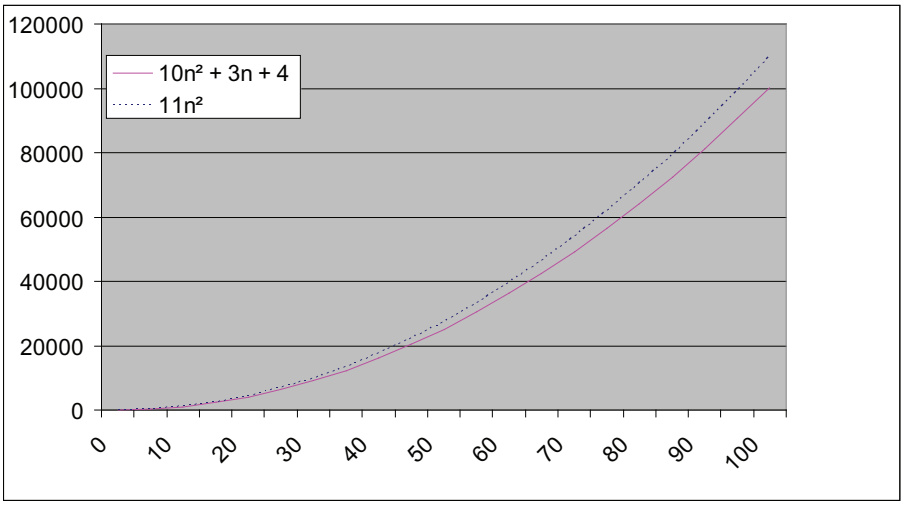


Figura 2 Gráfico mostrando que $11n^2$ (linha pontilhada) é um limitante superior para $10n^2 + 3n + 4$ (linha sólida).

Por outro lado, seria possível mostrar que $T(n) = 10n^2 + 3n + 4$ é $O(n)$? Para isso, é preciso mostrar que $10n^2 + 3n + 4 \leq cn$, para $n \geq n_0$. Não existem, porém, valores para c e n_0 que permitam esta demonstração. Assim, a afirmação que $T(n)$ pertence a $O(n)$ é falsa.

É também viável mostrar que $T(n) = 10n^2 + 3n + 4$ é também $O(n^3)$, assim como $O(n^4)$. Na realidade, pertence a $O(n^k)$, para qualquer k maior ou igual a 2. É claro que é mais significativo escolher a melhor $f(n)$, ou seja, a *menor* função que limita superiormente $T(n)$.

Para finalizar, deve-se notar que dadas as funções $T_1(n) = n^3 + 6n^2 + 10n + 5$ e $T_2(n) = 5n^3 + n^2 + 8n + 4$, é possível demonstrar que $T_1(n)$ é $O(T_2(n))$, assim como $T_2(n)$ é $O(T_1(n))$. Na verdade, ambas são $O(n^3)$, o que leva à conclusão que a notação mais simples é a mais adequada para dizer que ambas as funções possuem comportamento de crescimento segundo uma função cúbica. Assim, na escolha de uma função $f(n)$ para a notação $O(f(n))$, $f(n)$ deve ser dada na forma mais simples, isto é, sem constantes multiplicativas ou termos de n menos significativos.

A Figura 1 mostra as funções mais comuns usadas em algoritmos para indicar a complexidade. A ordem de apresentação começa pelas funções que indicam algoritmos menos complexos, aumentando sequencialmente sua complexidade. O comportamento dessas funções pode ser observado na Figura 3.

Tabela 1 Funções tradicionalmente escolhidas para indicar a complexidade de algoritmos, dadas em ordem crescente de complexidade.

Ordem relativa	Função
1	1
2	log *n*
3	*n*
4	*n*log *n*
5	n^2
6	n^3
7	2^n

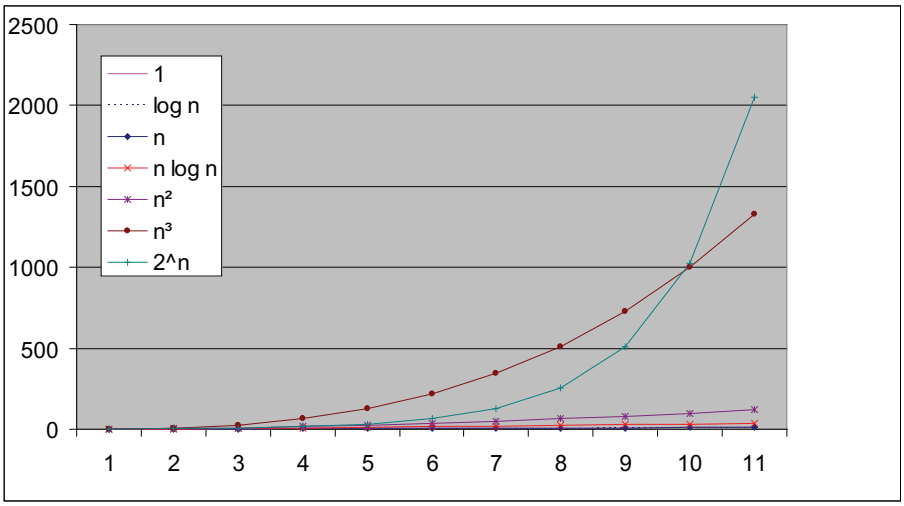


Figura 3 Comportamento das principais funções que descrevem a complexidade de algoritmos (veja Tabela 1).

1.3.2 O impacto da complexidade

Um ponto de grande importância da avaliação de algoritmos é a influência com que complexidades diferentes afetam o tempo de execução.

Um processador atual executa vários milhões de instruções por segundo. Isso quer dizer que um computador que execute um milhão de instruções por segundo leva o tempo de um microssegundo[3] (1μs) para cada instrução. Isso significa que, se um programa tiver tempo de execução pertencente a O(1), então ele levará 1 μs para ser executado. Na prática, ele leva tempo constante para execução (vários microssegundos), mas aqui se está fazendo apenas uma ilustração simplificada da situação, como se T(*n*) fosse apenas T(*n*) = 1.

3 Um microssegundo equivale a 10^{-6} segundo.

Essa simplificação será adotada para as demais complexidades exemplificadas. Desse modo, um programa que pertença a O(log *n*) levará 3,32 µs (ou seja, log 10^4) para executar para um tamanho de problema igual a 10. Para o mesmo tamanho de problema, um programa O(n^2) levará 100 µs, da mesma forma que um que fosse O(n^6) levaria 1.000.000 µs, ou seja, 1 segundo. Como outros exemplos, um programa O(*n* log *n*) terá tempo de execução de 4,48 ms (4.480 µs) quando *n* for igual a 500 e, para o mesmo tamanho de problema, um programa O(n^2) levará 250 ms, ou seja, vezes mais que o primeiro. Algoritmos de tempo exponencial, como O(2^n), são bastante sensíveis ao aumento no tempo de execução. Para *n* igual a 10, por exemplo, levará 1,2 ms, mas já para *n* igual a 20, passará a ter tempo de execução de 1,05 segundo. Bastou dobrar o tamanho do problema para o tempo de execução ficar vezes maior. Ou então, para *n* igual a 50, passará levar 13.031 dias, ou seja, 35,7 anos.

A Tabela 2 mostra os tempos de execução para vários algoritmos com diferentes ordens de complexidade, permitindo visualizar o peso com que o aumento no tamanho do problema influencia no tempo gasto para processamento.

Tabela 2 Exemplos de tempos de execução para algoritmos com ordens de complexidade variadas, supondo um computador que execute uma instrução por microssegundo.

n	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)
1	1,00 µs	< 1,00 µs	1,00 µs	0,00 µs	1,00 µs	2,00 µs
10	1,00 µs	3,32 µs	10,00 µs	33,22 µs	100,00 µs	1,02 ms
20	1,00 µs	4,32 µs	20,00 µs	86,44 µs	400,00 µs	1,05 s
30	1,00 µs	4,91 µs	30,00 µs	147,21 µs	900,00 µs	17,90 min
50	1,00 µs	5,64 µs	50,00 µs	282,19 µs	2,50 ms	13031,25 dias
100	1,00 µs	6,64 µs	100,00 µs	664,39 µs	10,00 ms	$4{,}02\times10^{13}$ séculos
500	1,00 µs	8,97 µs	500,00 µs	4,48 ms	250,00 ms	$1{,}04\times10^{131}$ milênios
1.000	1,00 µs	9,97 µs	1,00 ms	9,97 ms	1,00 s	$3{,}40\times10^{281}$ milênios
5.000	1,00 µs	12,29 µs	5,00 ms	61,44 ms	25,00 s	~∞
10.000	1,00 µs	13,29 µs	10,00 ms	132,88 ms	1,67 min	~∞
100.000	1,00 µs	16,61 µs	100,00 ms	1,66 s	166,67 min	~∞

Uma questão importante que surge dessa discussão é que simplesmente aumentar a velocidade de processamento de um computador não ajuda, caso o algoritmo implementado tenha uma complexidade alta. A Tabela 3 mostra o impacto do aumento da capacidade de processamento sobre o tamanho do problema que alguns algoritmos conseguem resolver em um dado período de tempo. Se o tempo

4 É importante lembrar que a notação **log** sempre utiliza a base 2 neste texto.

de execução for O(n) e o maior tamanho de problema que puder ser resolvido em um dado tempo for K, então em um computador 100 vezes mais rápido será possível resolver um problema de tamanho 100K. E em um supercomputador um bilhão de vezes mais rápido resolverá um problema 1.000.000.000 vezes maior, considerando o mesmo intervalo de tempo. Porém, para algoritmos O(n^2), se o computador for 100 vezes mais rápido, o problema resolvido será apenas 10 vezes maior para um mesmo intervalo de tempo. No supercomputador, será possível resolver um problema 31.623 vezes maior. Os algoritmos exponenciais, ou seja, aqueles O(2^n), possuem ganhos bem mais modestos. Assim, se em um dado período de tempo é resolvido um problema de tamanho K pelo computador atualmente disponível, se for comprado um computador 1000 vezes mais rápido, esse mesmo tempo será suficiente para resolver apenas um problema de tamanho K + 9,9. Isso quer dizer que, se o algoritmo resolver um problema para, por exemplo, 500 itens em 1 segundo, então no computador 1000 vezes mais rápido, 1 segundo resolverá apenas o problema para aproximadamente 510 itens; e o supercomputador usará seu 1 segundo para processar 530 itens.

Tabela 3 Tamanho do problema que seria executado em um período fixo de tempo, caso a velocidade de processamento fosse aumentada.

T(n)	Tamanho de problema resolvido	100 vezes mais rápido	1000 vezes mais rápido	1.000.000.000 vezes mais rápido
n	K	100 K	1000 K	10^9 K
n^2	K	10 K	31,6 K	31623 K
n^3	K	4,6 K	10 K	1000 K
n^5	K	2,5 K	3,9 K	63 K
2^n	K	K + 6,6	K + 9,9	K + 30

A conclusão óbvia é que, se um algoritmo tiver complexidade muito alta, simplesmente trocar o hardware não resolverá o problema, pois os ganhos podem não ser tão significativos. Assim se justifica o esforço para a elaboração de algoritmos melhores para a solução dos problemas.

1.3.3 O modelo de computador e os comandos

Todo computador consiste de pelo menos uma unidade de processamento, a memória principal e as unidades de entrada e saída. A unidade de processamento é capaz de executar os comandos do algoritmo e esse processamento leva tempo.

Pensando em um nível de detalhe relativamente grande, um comando como o mostrado no Algoritmo 2-1 desencadeia uma grande série de processamento. Primeiramente são feitos acessos à memória para recuperar os conteúdos das variáveis *a* e *b*; em seguida é realizada a soma e gerado um resultado intermediário (que pode ou não ter que ser armazenado temporariamente na memória); finalmente o resultado final é transferido para a memória no endereço correspondente á variável *c*. Logo, existem tempos para recuperação de dados da memória, para a soma e para o armazenamento do resultado na memória. Cada um destes tempos pode ser diferente, dependendo de como se definam as características do modelo de computador utilizado.

Algoritmo 2-1

```
1       c ← a + b
```

Como estes detalhes acabam não sendo relevantes para a análise dos algoritmos, muitas simplificações podem ser adotadas. Isso evita o esmero desnecessário de considerar que multiplicações são mais custosas que somas, ou que expressões aritméticas inteiras tendem a ser mais eficientemente tratadas que as de ponto flutuante.

Portanto, nas análises descritas aqui, cada operação, por mais diferente que seja, será considerada como consumindo exata uma unidade de tempo. Isso leva a uma forma simples de contar tempo, fazendo que o comando do Algoritmo 2-1 consuma 4 unidade de tempo (ut): duas recuperações de dados, uma soma e uma atribuição.

Ainda assim, à medida que os conceitos da análise forem sendo entendidos, simplificações ainda maiores serão feitas. Dessa forma, preocupar-se com a exatidão dos cálculos na contagem das instruções que serão exemplificadas a seguir é desnecessário, pois o que interessa é a “cara” da função de tempo, e não os valores dos coeficientes que a compõem.

A abordagem dos tempos de processamento será considerada, na sequência, por suas diferentes apresentações, pois os algoritmos não são analisados diretamente como um todo, mas por partes.

1.3.4 Trechos de tempo constante

O Algoritmo 2-2 mostra uma solução para o seguinte problema: Dado o valor de um número inteiro positivo, calcular e apresentar a soma de todos os inteiros positivos menores ou iguais a ele.

Algoritmo 2-2

```
1     { soma de positivos até um dado valor }
2     algoritmo
3           declare valor, soma: inteiro
4
5           leia(valor)
6           soma ← valor * (valor + 1)/2
7           escreva(soma)
8     fim-algoritmo
```

Este algoritmo, para ser analisado, precisa inicialmente ter determinado o que é seu tamanho de problema. Verificando o código, é possível verificar que, independente do valor digitado pelo usuário, o algoritmo consome exatamente o mesmo tempo.

Considerando-se que o comando *leia* (linha 1.3.4) consuma 2 ut (pela leitura em si e pelo armazenamento na variável), a linha 1.3.4 leve 6 ut (duas recuperações, uma soma, uma multiplicação, uma divisão e uma atribuição) e na linha 1.3.4 a escrita consuma outras 2 ut, uma para recuperar o conteúdo de *soma* e outra para fazer a saída para a tela, o tempo de execução do algoritmo é sempre T(n) = 10 ut.

Para os comandos desse algoritmo, não existe tamanho do problema, pois não há variação de tempo em função de nada.

Embora isso ocorra, neste exemplo, para todos os comandos do algoritmo, a análise é similar para trechos de algoritmo que contenham apenas leituras, escritas e atribuições com expressões simples. Esses trechos consomem tempo constante, sendo que a influência do tamanho do problema nestes trechos inexiste. Nestas situações, é comum escrever que estes trechos possuem T(n) = k, sendo k uma constante.

Quando o tempo de execução é constante, é fácil mostrar que T(n) é O(1)[5]. Para T(n) = k, basta escolher $c = k$ para demonstrar que T(n) $\leq c^{3}2$, pv t f lb, r vf $k \leq k$, fato verdadeiro para qualquer valor escolhido para n_0.

5 A função mais simples para ser usada na notação O(f(n)) é f(n) = 1, que indica o tempo de execução constante.

1.3.5 Trechos com repetições

Dentre os comandos de algoritmos que geram consumo de tempo, as repetições são as principais. Quanto maior o número de repetições, mais vezes os comandos são executados e, portanto, maior o tempo consumido.

Para exemplificar, vamos considerar novamente o problema da soma dos valores de 1 até um valor inteiro positivo dado pelo usuário. O Algoritmo 2-3 mostra outra solução para o mesmo problema.

Algoritmo 2-3

```
1     { soma de positivos até um dado valor ]
2     algoritmo
3           declare i, valor, soma: inteiro
4
5           leia(valor)
6           soma ← 0
7           para i ← 1 até valor faça
8                 soma ← soma + i
9           fim-para
10          escreva(soma)
11    fim-algoritmo
```

As linhas 1.3.5, 1.3.5 e 1.3.5 não dependem de nada e consomem, respectivamente, 2, 1 e 2 ut. Na repetição das linhas 1.3.5 a 1.3.5 a situação é diferente, pois o dado digitado pelo usuário para a variável *valor* influencia no número de repetições realizadas. Neste caso, o valor digitado é o tamanho do problema, ou seja, *n*. O comando *para* tem uma iniciação para *i* receber o valor 1 (1 ut), mais *n* incrementos (3 ut cada um), além de *n* + 1 comparações para verificar se a repetição terminou (uma antes de iniciar e uma para cada incremento). Cada comparação consulta o valor de *i* e o compara a *valor* (consumindo 3 ut). O comando da linha 1.3.5 consome 4 ut e é executado *n* vezes.

Assim, no final, o algoritmo como um todo consome $T(n) = 10n + 9$. Em outras palavras, se o valor digitado for 10, então o tempo de execução será de 109 unidades de tempo; se for 55, T(55) = 559 ut; se for 0, T(0)= 9 (leitura, atribuição

a *soma*, atribuição inicial para *i*, uma verificação do *para* ao determinar que não haverá execução e a escrita do resultado).

Claramente, tanto o Algoritmo 2-2 quanto o Algoritmo 2-3 produzem o mesmo resultado. Porém o primeiro é melhor que o segundo para qualquer *n* exceto 0.

Em termos de complexidade, o Algoritmo 2-2 tem tempo constante e é, portanto, O(1). Já o Algoritmo 2-3 é O(*n*). Recorrendo à Tabela 1 torna-se simples observar que o primeiro algoritmo é superior ao segundo, considerando os tempos de execução para valores grandes de *n*, ou seja, em termos de complexidade assintótica.

A partir de agora, outros exemplos são considerados, mas apenas trechos de código serão apresentados. Independendo do contexto de um problema em si, apenas os comandos são analisados.

O código apresentado no Algoritmo 2-4 considera que o valor da variável **n** represente o tamanho do problema. Neste caso, **n** e *n* são os mesmos.

Algoritmo 2-4

```
1     leia(n)
2     para i ← 1 até n/2 faça
3           escreva(i)
4     fim-para
```

O número de repetições, neste código, é a metade do tamanho do problema. Assumindo *n* par, o código consome 2 ut para a leitura, 1 ut para iniciar *i*, realiza *n*/2 + 1 comparações de término (3*n*/2 + 1 ut), faz *n* incrementos (3*n*/2 ut) e executa o comando *escreva* *n*/2 vezes (*n* ut). Assim, o tempo consumido por este trecho é T(*n*) = 4*n* + 6.

Claramente o Algoritmo 2-4 tem tempo de execução pertencente a O(*n*). Em termos práticos, isso quer dizer que seu comportamento é similar ao do Algoritmo 2-3, também O(*n*). É importante notar que o *comportamento* é similar, e não os tempos em si, dado que o tempo de execução do Algoritmo 2-3 é mais que o dobro do Algoritmo 2-4, para praticamente qualquer valor de *n*.

É importante notar que, para o Algoritmo 2-4, mesmo que a repetição tivesse seu limite final modificado de **n**/2 para qualquer outro valor **n**/*k*, qualquer *k* maior que 1, o tempo de execução ainda seria O(*n*), pois T(*n*) continuaria a ser uma função linear, na forma *an* + *b*.

O Algoritmo 2-5 mostra o aninhamento de repetições.

Algoritmo 2-5

```
1     leia(n)
2     para i ← 1 até n faça
3           para j ← 1 até n faça
4                 escreva(i * j)
5           fim-para
6     fim-para
```

A análise deste trecho exige que as repetições sejam consideradas separadamente, iniciando-se com a mais interna. Para a repetição das linhas 1.3.5 a 1.3.5, considerada independentemente do *para* mais externo, consome os seguintes tempos: 1 ut para iniciação de *j*, 3*n* ut para os *n* incrementos, 3(*n* + 1) ut para as *n* + 1 comparações de término e, finalmente, 4*n* ut para o comando *escreva* e seus cálculos. O tempo de cada *para* relativo à variável *j* tem, portanto, $T_j(n) = 10n + 4$ ut.

A repetição iniciada na linha 1.3.5 consome: 1 ut para a iniciação de *i*, 3*n* ut para os incrementos, 3(*n* + 1) ut para os testes de término e $nT_j(n)$ pelas *n* execuções do *para* em *j*. Desse modo, o tempo da repetição das linhas 1.3.5 a 1.3.5 tempo de execução $T_i(n) = 4n + 4 + nT_j(n)$ ut, ou seja, $T_i(n) = 4n + 4 + n(10n + 4)$ e, portanto, $T_i(n) = 10n^2 + 8n + 4$ ut.

Acrescentando-se, finalmente, as 2 ut do comando *leia*, o tempo total do trecho do Algoritmo 2-5 é $T(n) = 10n^2 + 8n + 6$ ut. Desse modo, para *n* igual a 10, T(10) = 1086 ut e para um valor 2 vezes maior, T(20) = 4166 ut. Dobrando-se o tamanho do problema, o tempo de execução ficou 3,84 vezes maior, praticamente quadruplicando o tempo. O consumo de tempo em repetições é importante e assume proporções ainda mais significativas quando há aninhamentos. Para este algoritmo, a complexidade pertence a $O(n^2)$, ou seja, o comportamento de tempo em relação ao tamanho do problema, é quadrático.

Considerando-se agora o trecho de código apresentado no Algoritmo 2-6 (que tem repetição similar à usada para fazer a transposição de elementos de uma matriz quadrada), deve-se notar que o número de repetições do laço interno varia.

Algoritmo 2-6

```
1     leia(n)
2     para i ← 1 até n faça
3           para j ← i até n faça
4                 escreva(i * j)
5           fim-para
6     fim-para
```

Para entender o número total de repetições, opta-se aqui por contá-las, para cada uma das repetições do laço em *i*. A repetição *para* em *i* executa exatamente *n* vezes, enquanto a repetição em *j* depende de cada valor de *i*. Assim, na primeira vez, o laço interno executa *n* vezes. Na segunda repetição, a execução é de *n* – 1 vezes. Sequencialmente, cada nova execução do *para* em *j* executa uma vez a menos, sendo que na última repetição é feita apenas uma vez. Podem ser contadas, para começar, *n* iniciações de *j*, consumindo 3*n* ut. O número total de incrementos de *j* é de *n* (quando *i* vale 1) decrescendo até 1 (quando *i* vale *n*), ou seja, $n + (n\text{-}1) + (n\text{-}2) + ... + 1$, o que resulta em $n(n + 1)/2$ vezes, consumindo $3n(n + 1)/2$ ut. Para os incrementos o raciocínio é similar, começando com $n + 1$ e terminando com 2. Portanto, são $(n+1) + n + (n\text{-}1) + (n\text{-}2) + ... + 2$, totalizando $(n^2 + 3n)/2$ vezes, ou $3(n^2 + 3n)/2$ ut. Dentro da repetição, o comando *escreva* é executado $n(n + 1)/2$ vezes, consumindo $2n(n + 1)$ ut.

Para fechar, existem 2 ut da leitura inicial e 1 ut da iniciação da variável *i*. São também 3*n* ut para os incrementos de *i* e outras $3(n + 1)$ para as verificações das condições de término. O total geral para este código é, finalmente, $T(n) = 5n^2 + 17n + 6$ ut. Grosseiramente, pode-se dizer que equivale à metade do tempo quando comparado às repetições do Algoritmo 2-5, porém, sendo também uma função quadrática, com comportamento $O(n^2)$.

De forma geral, as repetições que possuem tanto *incrementos constantes* como *número total de repetições como uma multiplicação do tamanho do problema por uma constante* são avaliadas como $O(n)$. Os comandos apresentados no Algoritmo 2-7 são exemplos desta situação, desde que se considere que os comandos internos (omitidos aqui) possuam tempo constante. Em todos os exemplos a variável **n** representa o tamanho do problema *n*.

Algoritmo 2-7

```
1     para i ← 1 até n faça
2     para i ← 1 até n/2 faça
3     para i ← 1 até n/10 faça
4     para i ← 1 até n/k faça  { k constante }
5     para i ← 1 até 10 * n faça
6     para i ← 1 até k * n faça { k constante }
7     para i ← n/2 ate 5 * n faça
8     para i ← 1 até k1 * n + k2 faça { k1 e k2 constantes }
9     i ← 1
10    enquanto i < n faça
11          { algum processamento de tempo constante }
12          i <- i + k  { k constante }
13    fim-enquanto
```

De forma similar, aninhamentos de repetições com essas mesmas características apenas aumentam a ordem do polinômio, sendo que duas repetições representam O(n^2), três repetições O(n^3) e assim por diante. O Algoritmo 2-8 mostra alguns exemplos.

Algoritmo 2-8

```
1     para i ← 1 até n faça          { O(n²) }
2           para j ← 1 até n faça
3     para i ← 1 até n/k faça        { O(n²) }
4           para j ← 1 até n faça
5     para i ← 1 até n faça          { O(n²) }
6           para j ← 1 até n * k faça
7     para i ← 1 até k1 * n faça     { O(n³) }
8           para j ← 1 até n/k2 faça
9                 para k ← j até n faça
10    para i ← 1 até n faça          { O(n³) }
11          para j ← 1 até n-i faça
```

```
12                para k ← i até j faça
13    para i ← 1 até n faça          { O(n³) }
14          para j ← i até n faça
15                k ← j
16                enquanto k > i faça
17                      { algum processamento constante }
18                      k ← k - 1
19                fim-enquanto
```

Essas considerações aplicam-se, portanto, se estas duas condições ocorrerem:

1. O número total de repetições é linearmente proporcional a *n*.
2. O incremento é constante para a variável de controle da repetição.

Porém, nos casos em que a variável de controle do laço de repetição não tem incrementos constantes, mas é multiplicada ou dividida por uma constante, essa regra não se aplica.

Nessas situações específicas, outro tipo de comportamento emerge da repetição e um exemplo auxilia no entendimento. O código apresentado no Algoritmo 2-9 tem como tamanho do problema o valor lido para a variável **n**, considerada inteira A cada repetição, seu valor é reduzido à metade, de forma que, se o valor digitado for 16, as saídas serão 16, 8, 4, 2 e 1 para cada uma das repetições.

Algoritmo 2-9

```
1     leia(n)
2     enquanto n ≥ 1 faça
3           escreva(n)
4           n ← n/2
5     fim-enquanto
```

Esse comportamento não pode ser considerado O(*n*), pois o controle da repetição (no caso a variável **n**) não é incrementado ou decrementado por um valor constante, mas sim dividida por um valor constante (2).

A avaliação do tempo de execução desse algoritmo tem que considerar o número de repetições que serão realizadas. Assumindo que esse valor seja k (a ser determinado depois), a contagem do custo de tempo pode ser feita da seguinte forma: 2 ut na leitura e armazenamento da linha 1.3.5; $k + 1$ comparações do *enquanto* (linha 1.3.5), o que consome $2(k + 1)$ ut; os comandos nas linhas 1.3.5 e 1.3.5 consomem 5 ut para cada vez, totalizando $5k$ ut de tempo no total. $T(n)$, por enquanto, pode ser escrito como $7k + 4$. Mas ainda falta determinar o valor de k em função de n.

Supondo, para facilitar, que o valor digitado seja uma potência de 2, escrita 2^m, o Algoritmo 2-9 produzirá as saídas 2^m, 2^{m-1}, 2^{m-2}, 2^{m-3}, ..., 2^2, 2^1 e 2^0. É fácil deduzir que a repetição é executada $m + 1$ vezes. Assim, se k é o número de repetições, então $k = m + 1$ e $T(n) = 7m + 11$.

Finalmente, se o valor lido é o tamanho do problema n, então o tamanho do problema pode ser escrito como 2^m, que é o valor efetivamente lido. Portanto, $n = 2^m$, ou seja, $m = \log n$. Isso permite reescrever a expressão do tempo de execução, obtendo-se $T(n) = 7\log n + 11$, o que é uma repetição $O(\log n)$.

Ainda considerando que a variável **n** seja inteira, mas o valor lido não seja uma potência de 2, o raciocínio da análise se mantém apenas com valores minimamente diferentes para as constantes obtidas, mas ainda $T(n)$ é na forma $T(n) = a\log n + b$. Ou seja, repetição continua sendo $O(\log n)$.

Supondo agora que o valor constante da divisão (linha 1.3.5) não seja 2, mas 3, o raciocínio pode ser repetido considerando-se uma entrada 3^m, o que leva a se obter uma função de tempo na forma $T(n) = a\log_3 n + b$. Se o valor de **n** fosse dividido por 7, a função seria na forma $T(n) = a\log_7 n + b$. Para qualquer que seja a situação, a função do tempo de execução é uma função logarítmica, apenas com base diferente: $T(n) = a\log_p n + b$.

Usando-se o recurso de conversão de base, para converter $T(n) = a\log_p n + b$ para base 2, obtém-se $T(n) = a(\log n/\log p) + b$, pois $\log_w x$ é igual a $\log_z x/\log_z w$. Como $\log p$ é um valor constante, é possível escrever $a/\log p$ como a_1, de forma que $T(n) = a_1\log n + b$, o que leva a concluir que, apenas por diferenças nas constantes que multiplicam a parte do logaritmo, qualquer que seja o valor da divisão aplicada à variável de repetição, a repetição como um todo é sempre $O(\log n)$.

Pelo uso de uma abordagem similar, não apresentada neste texto, é também viável concluir que, caso a variável de controle do laço de repetição seja multiplicada por um valor constante, a repetição terá custo também $O(\log n)$, como é o caso dos comandos do Algoritmo 2-10. Novamente, o valor usado na multiplicação não é importante, desde que seja constante.

Algoritmo 2-10

```
1     leia(n)
2     i ← 1
3     enquanto i < n faça
4           escreva(i)
5           i ← i * 2
6     fim-enquanto
```

Dessa forma, uma repetição é avaliada como O(log *n*) quando:

A comparação de término da repetição é feita em relação a um valor que seja uma variação linear de *n*.

A variável de controle é multiplicada ou dividida por um valor constante.

Outro exemplo, apresentado no Algoritmo 2-11, mostra a combinação de dois laços de repetição. O mais externo, feito pelo comando *enquanto*, usa a variável *i* para controle da repetição, a qual é multiplicada por 5 a cada repetição. A repetição mais interna, definida pelo comando *para*, usa incrementos constantes (de 1 em 1) para a variável *j*.

Algoritmo 2-11

```
1     leia(n)
2     i ← 1
3     enquanto i < n faça
4           para j ← 1 até n faça
5                 escreva(i, j)
6           fim-para
7           i ← i * 5
8     fim-enquanto
```

Avaliando o algoritmo, é possível, mesmo sem contar o custo específico das instruções, visualizar que o *enquanto* executa os comandos internos (linhas 1.3.5 a 1.3.5) um número de vezes proporcional a log *n*. O custo do comando *para*, individualmente, é proporcional a *n*. O comando mais executado é o *escreva* da linha 1.3.5, o qual é executado um número de vezes proporcional a log *n*

vezes n, ou seja, $n\log n$. T(n) pode ser avaliado, grosseiramente, como T(n) = $an\log bn + cn + d\log n + e$. A conclusão é que T($n$) pertence a O($n\log n$), por fim.

Todas as análises de algoritmos com laços de repetição devem ser consideradas sobre avaliações de quantas repetições são feitas, permitindo que se estime a forma da função de tempo que descreve o custo computacional relativo ao tempo. Algumas dicas ajudam, mas somente uma consideração mais detalhada pode precisar a ordem de complexidade efetiva.

1.4 Outras considerações

A análise de algoritmos apresentada leva em consideração o comportamento dos algoritmos para grandes tamanhos de problemas. Entretanto, isso não significa que um algoritmo pertencente a O(n^2) seja sempre pior que um O(n), já que esta é uma avaliação geral.

Assim, tomando-se como exemplo $T_1(n) = 1000n + 100$ e $T_2(n) = 2n^2 + 5n + 1$, sabe-se que $T_1(n)$ é O(n) e $T_2(n)$ é O(n^2). Em princípio, o primeiro é superior ao segundo, dado que sua taxa de crescimento é linear, enquanto a do outro é quadrática.

Porém, caso seja necessário utilizar os algoritmos para um número mais reduzido de dados, o segundo passa a levar vantagem sobre o primeiro. Para um tamanho de problema aproximadamente igual a 500, o tempo de execução do algoritmo O(n^2) é melhor, pois leva menos tempo. Para valores maiores, esta vantagem cessa e o primeiro passa a ser consideravelmente mais rápido.

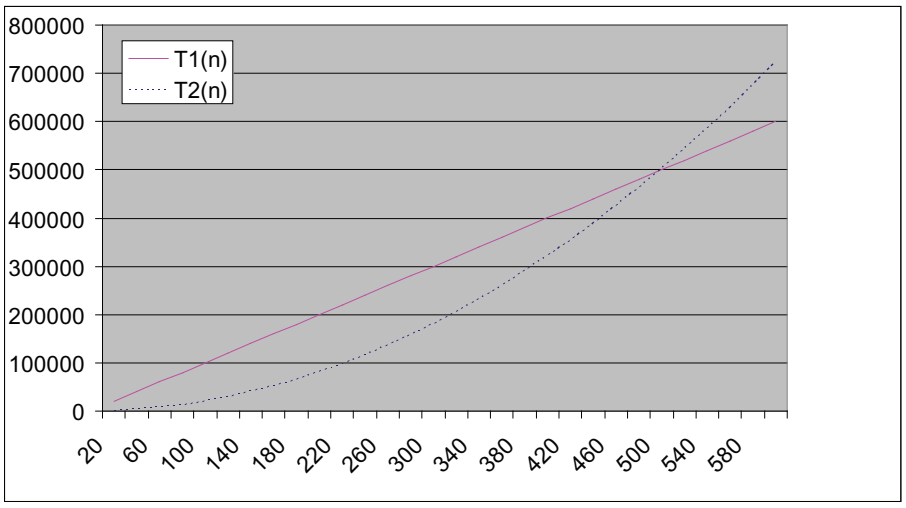


Figura 4 Gráfico com os tempos de processamento de dois algoritmos de ordens de complexidade diferentes.

Esta situação é verdadeira para um grande número de situações. Muitas vezes um algoritmo que não se comporta bem para grandes volumes de dados, por ter sido escrito sem grandes sofisticações de estratégia, pode se comportar melhor que outro mais sofisticado. Muitas vezes, os algoritmos mais sofisticados (que melhoram a complexidade por empregar estratégias mais elaboradas) obtêm seu sucesso à custa de uma preparação inicial dos dados. Entretanto, quando há poucos dados, pode ser bem mais rápido usar um processamento simples do que perder tempo fazendo uma organização complexa para um se obter um processamento posterior simplificado.

1.5 Considerações finais

A análise de algoritmos envolve determinar, a partir da avaliação do código e estrutura dos comandos, qual a principal função que descreve o custo de tempo de execução do algoritmo.

A notação assintótica visa estabelecer um padrão para o qual se permitem comparações entre algoritmos. Assim, havendo dois ou mais algoritmos para resolver um determinado problema, é possível realizar considerações com melhor embasamento quando se decide adotar um ou outro. As avaliações dos tempos de execução também não consideram importantes as constantes multiplicativas que envolvem as funções T(n), mas apenas o comportamento geral que ela apresenta.

Naturalmente, o tempo de execução de um algoritmo pode ser obtido na prática. É viável fazer a implementação do algoritmo em uma linguagem, compilar o programa e executá-lo, medindo os tempos de execução. Essa solução prática, porém, envolve fatores delicados, como, por exemplo, comparar implementações que executam em computadores diferentes. Neste aspecto, implementações sofrem influências de vários fatores, como:

- Linguagem de programação escolhida;
- Compilador utilizado para a geração do código;
- Opções de otimização que cada compilador possui;
- Programador que codificou o algoritmo;
- Processador usado na execução e suas características.

Para encerrar, é importante lembrar que a análise fornece uma visão da taxa de crescimento que o tempo de execução do algoritmo possui. Observar somente este resultado é uma opção arriscada, pois outros fatores (como o volume de dados processados) podem interferir no tempo real necessário. Além disso, há algoritmos O(n) muito melhores ou muito piores que outros algoritmos O(n).

UNIDADE 2

Métodos de ordenação

2.1 Primeiras palavras

A utilização dos dados segundo um critério de ordenação auxilia bastante na execução de várias tarefas. A ordenação alfabética de um índice remissivo é essencial para que a localização de um termo seja eficiente. A ordenação está presente em listas telefônicas, ordenadas alfabeticamente por sobrenome e, para um mesmo sobrenome, ordenadas alfabeticamente por nome; ou ainda os livros na biblioteca, ordenados por código de classificação.

Para que qualquer ordenação possa ser feita, é preciso que se defina um critério para ela. Este critério exige a definição do que será usado para a ordenação e qual a relação de desigualdade existente. Para valores numéricos o critério é a relação usual, dada pela magnitude dos números, de forma que valores menores vêm antes dos maiores. Esse critério determina, por exemplo, que pessoas mais novas antecedem as mais velhas, definindo um modo natural de ordenação. Para o caso de nomes, a ordem alfabética é a mais comum, de maneira que a ordem definida pelo alfabeto é usada como critério para a ordem dos nomes. Para o caso dos ideogramas chineses, a ordenação se dá conforme a complexidade de cada ideograma em termos do número de traços que o compõe.

A tarefa de colocar em ordem é, portanto, importante e a eficiência com que isso é feito, relevante em muitos contextos.

Para a utilização de algoritmos para abordarem o problema da ordenação, é essencial que sejam visualizadas duas situações: os métodos para memória principal e os para memória secundária. Dadas as diferentes características entre os dois tipos de memória, as abordagens para efetuar a ordenação em si usam estratégias distintas.

Um último ponto é que, embora importantes, os métodos de ordenação estão amplamente disponíveis na literatura e também *on-line*. Este texto, portanto, não tem o objetivo de descrevê-los a fundo, mas de caracterizá-los segundo seus comportamentos e aplicações.

2.2 Ordenação na memória principal

Para os métodos em memória principal, uma característica interessante que todos possuem é que os dados são permutados entre si, não exigindo espaço extra para a transferência de dados, exceto o preciso para efetuar trocas individuais. Esta característica se contrapõe aos métodos usados para ordenação em memória secundária, na qual é mais interessante fazer cópias de todo o conjunto de dados, mas isso será abordado na próxima sessão.

Nos diversos exemplos, a ordenação será discutida sobre arranjos com valores numéricos, apenas por serem mais simples de visualizar e por serem mais compactos que nomes. É claro que, dado um critério qualquer de ordenação, basta adaptar para que as comparações obedeçam a este critério. Ordens inversas, isto é, do maior para o menor, podem ser definidas pela alteração destes critérios.

Para ver os métodos de ordenação mais conhecidos, eles estão divididos em dois grupos: os básicos e os avançados. Os métodos de ordenação básicos são os mais elementares e, em geral, menos eficientes. Os avançados incluem os que manipulam os dados com estratégia mais sofisticada, sendo estes mais eficientes para grandes volumes de dados.

2.2.1 Métodos básicos de ordenação

Neste item são abordados três métodos para ordenação que usam estratégias diferentes. Eles são denominados *inserção direta*, *seleção direta* e *bubblesort*.

O método de *inserção direta*, também chamado de ordenação por inserção, utiliza uma estratégia muito simples. Inicialmente, é separado o primeiro elemento do conjunto; em seguida, um novo elemento é pego e posto em ordem em relação ao primeiro. A cada novo elemento, este é colocado já em ordem em relação ao grupo já ordenado, repetindo-se o processo até o último elemento.

Para a aplicação dessa estratégia em arranjos, segue os seguintes passos:

1. Seleciona-se o primeiro elemento do arranjo, o qual define o subconjunto ordenado inicial.
2. São pegos os demais elementos um a um, a partir do segundo, e posicionados em relação à parte ordenada, fazendo os deslocamentos de dados necessários

Algoritmo 3-12

```
1     inserção_direta(dados[]: inteiro; tamanho: inteiro)
2     para i ← 1 até tamanho – 1 faça
3           { seleciona um dos elementos }
4           auxiliar ← dados[i]
5
6           {abre espaço deslocando os elementos maiores que dados[i]}
```

```
7           j ← i - 1
8           enquanto dados[j] > auxiliar e j ≥ 0 faça
9                 dados[j + 1] ← dados[j]  { desloca uma posição }
10                j ← j - 1
11          fim-enquanto
12
13          { coloca na posição correta }
14          dados[j + 1] ← auxiliar
15    fim-para
```

A inserção direta é descrita em sua forma algorítmica no Algoritmo 3-12. Deve-se notar que a varredura pelos elementos se inicia no segundo elemento (posição 1 do vetor[6]), já que o elemento da posição zero já é considerado ordenado (quando há um único elemento no conjunto, o conjunto pode ser considerado ordenado). A partir do primeiro, cada um dos elementos é posicionado em relação aos demais, fazendo os deslocamentos de todos os valores maiores que o elemento que está sendo ajustado. O espaço "aberto" no arranjo permite a inserção do elemento na sua devida posição.

O método é chamado de inserção por pegar cada elemento e o *inserir* na parte ordenada do arranjo.

A análise desse algoritmo passa, inicialmente, por identificar que o tamanho do problema *n* é dado pelo número de elementos presentes no vetor. Quanto maior o número de elementos, mais repetições são necessárias tanto para o comando *para* (linha 2.2.1) quanto para o *enquanto* (linha 2.2.1). O segundo ponto é notar que existe também uma dependência dos dados, a qual ocorre no comando *enquanto*. Neste comando, deve-se observar que a quantidade de elementos que são deslocados depende do valor do elemento; se ele for maior que a grande parte dos demais, poucos elementos devem ser movidos; se for o menor de todos, haverá movimentação de todos estes para que o elemento atual seja colocado na posição correta. Assim, para este algoritmo, existe tanto casos *melhor* e *pior* como também de caso *médio*.

Para seu *melhor* caso, o método inserção direta faz poucos movimentos, situação que ocorre quando o vetor original já se encontra praticamente ordenado. Para este caso, pode-se considerar que a repetição interna (comando

6 Lembra-se, aqui, que os dados nos arranjos são representados com a primeira posição sendo considerada a de índice zero.

enquanto) possui tempo constante; isto é, não depende do número total de elementos, já que repete o *enquanto* poucas vezes. Portanto, somente a repetição do *para* é relevante, fazendo uma única passagem pelos dados. Para o melhor caso, o algoritmo se comporta como O(*n*).

O pior caso ocorre quando os elementos estão originalmente em ordem decrescente, ou quase nesta situação. Neste cenário, o comando *enquanto* repete praticamente o número máximo de vezes, pois todos os dados têm que ser deslocados para cada novo elemento que é ordenado. Dessa forma, a repetição do *enquanto* é proporcional ao número de elementos do arranjo. Considerando a repetição do *para* e a do *enquanto*, o algoritmo torna-se pertencente a O(n^2).

Finalmente, existe a situação do caso médio, que considera que os dados estejam em uma ordem aleatória qualquer, ou seja, nem ordenados nem inversamente ordenados. Para este quadro, é possível considerar que, em média, a repetição do *enquanto* desloca metade dos itens do conjunto já ordenada ao inserir um novo elemento. É possível, assim, considerar que exige metade do esforço computacional para esta repetição, comparada ao que exige o pior caso. Mesmo assim, essa repetição é proporcional a *n* e, portanto, o algoritmo também é O(n^2) para o caso médio.

O importante, em geral, é o caso médio, pois se aplica a uma solução genérica. O pior caso é uma situação extrema, assim como o melhor caso. O conhecimento do bom comportamento, O(*n*), para o melhor caso passa a ser importante em alguns casos, quando já se sabe que há certa ordenação no conjunto de dados. De fato, um dos algoritmos avançados, o *shellsort*, faz uma organização dos dados para fazer uma "pré-ordenação" geral e depois utiliza dessa vantagem da inserção direta para finalizar a ordenação, obtendo com isso um excelente desempenho.

Usando uma estratégia diferente, o método conhecido por *seleção direta* (ordenação por seleção ou *selection*) é outro método básico. Seu princípio de operação é muito simples: o menor elemento deve ocupar a primeira posição; o segundo menor elemento deve ocupar a segundo; e assim por diante.

Sua estrutura básica reflete:

1. Considere todo o arranjo como não ordenado.
2. Para a parte do arranjo ainda não ordenada, localize o menor elemento e o mova para a posição, reduzindo o tamanho da partição não ordenada.

Algoritmo 3-13

```
1     seleção_direta(dados[]: inteiro; tamanho: inteiro)
2           para i ← 0 até tamanho – 2 faça
3                 { localiza o menor elemento da partição }
4                 menor ← dados[i]
5                 posição ← i
6                 para j ← i + 1 até tamanho – 1 faça
7                       se dados[j] < menor então
8                             menor ← dados[j]
9                             posição ← j   { armazena a posição }
10                      fim-se
11                fim-para
12
13                { coloca o menor elemento no início da partição }
14                dados[posição] ← dados [i]
15                dados[i] ← menor
16          fim-para
```

A estrutura básica da abordagem por seleção é apresentada no Algoritmo 3-13. Note-se que, a cada passo, é feita a seleção do elemento que deve ocupar cada posição.

Em termos de análise, pode-se notar que não existe dependência de nenhuma repetição em relação ao valor dos dados. Isso quer dizer que não há casos pior, médio ou pior, pois o comportamento é igual qualquer que seja a distribuição inicial dos dados no arranjo. A repetição do *para* mais externo (linha 2.2.1) executa sempre n vezes, enquanto a do interno (linha 2.2.1) se inicia com $n – 1$, depois passa por $n – 2$, $n – 3$ e assim por diante, até 2. Ambos os laços de repetição, como já foi visto, são proporcionais a n e o algoritmo tem comportamento $O(n^2)$. Desta maneira, pode-se considerar que seu desempenho é conhecido, pois não depende dos dados.

Por final, existe um algoritmo elementar amplamente apresentado nos cursos de computação: o chamado *bubblesort* ou método das bolhas (ou borbulhamento). A estratégia de ordenação consiste em varrer os dados do arranjo, do fim para o início e, sempre que encontrado um par de elementos adjacentes

fora de ordem, fazer a troca. Após um número de passadas pelo arranjo, a ordenação será obtida. Na primeira varredura, o menor elemento migrará para a primeira posição; a cada passagem subsequente, o próximo menor elemento migra para sua posição.

Sua estratégia é descrita por:

1. Faça a varredura, do fim ao início do arranjo, trocando os itens adjacentes que estejam fora de ordem.
2. Esta mesma estratégia é aplicada sucessivamente, até que a ordenação seja obtida.

Algoritmo 3-14

```
1     bubblesort(dados[]: inteiro; tamanho: inteiro)
2           para i ← 0 até tamanho – 2 faça
3                 { varredura }
4                 j ← tamanho – 1
5                 enquanto j > i faça
6                       se dados[j] < dados[j - 1] então
7                             { troca se forem adjacentes }
8                             auxiliar ← dados[j]
9                             dados[j] ← dados[j – 1]
10                            dados[j – 1] ← auxiliar ]
11                      fim-se
12                      j ← j – 1
13                fim-enquanto
14          fim-para
```

Como característica de funcionamento, pesa para o *bubblesort* o fato de o número de trocas poder ser elevado. Isso acontece, por exemplo, quando o menor elemento do arranjo está na última posição. Na primeira varredura, é feita sua troca com o elemento da posição anterior (exigindo três movimentações entre variáveis – linhas 2.2.1 a 2.2.1); decrementando *j*, nova troca é feita (mais três movimentos); e isso ocorre para cada valor de *j* na primeira varredura.

A análise do algoritmo do *bubblesort* permite verificar que o número total de repetições é fixo. O laço mais externo do *para* repete $n - 1$ vezes; a repetição interna, comando *enquanto*, repete uma vez a menos cada vez, mas é também proporcional a n. O algoritmo como um todo é pertencente a $O(n^2)$. O importante a notar é que, embora tenha mesma complexidade dos anteriores, isso significa apenas que seu comportamento é limitado por uma quadrática quando n cresce. Dado o custo das trocas necessárias, seu desempenho costuma ser muito inferior aos demais métodos.

2.2.2 Métodos avançados de ordenação

Para as estratégias mais sofisticadas de ordenação para memória principal serão considerados três métodos: *shellsort*, *heapsort* e *quicksort*. Cada um deles possui uma estratégia distinta e, também, características distintas.

O *shellsort* é um método inspirado na estratégia do inserção direta e se aproveita de seu melhor caso, que é quando o arranjo está praticamente ordenado. Ao invés de comparar elementos adjacentes, as comparações do *shellsort* são feitas entre elementos relativamente distantes entre si. Essa distância entre os elementos é chamada *incremento*. Assim, para um incremento valendo 5, as comparações são feitas entre os elementos das posições 0, 5, 10, etc.; das posições 1, 6, 11, etc.; das posições 2, 7, 12, etc., e assim por diante. Cada um destes subconjuntos é chamado *subsequência*. Na prática, para um dado valor de incremento p, são criadas p subsequências, cada uma com (aproximadamente) n/p elementos em cada uma.

A estratégia geral do *shellsort* pode ser dada por:

1. Defina alguns incrementos diferentes p_k ($p_k > p_{k+1}$) e, para cada incremento ordene cada subsequência usando a estratégia do método inserção direta.
2. Aplique, como última etapa, o inserção direta para garantir a ordenação completa.

Para cada incremento diferente, as várias subsequência são ordenadas de forma individual. Uma vantagem dessa abordagem é a movimentação de dados a “distâncias” razoáveis dentro do arranjo. Por exemplo, para um incremento igual a 43, as trocas ocorrerão diretamente entre as posições i e $i - 43$. Para um dado valor de incremento, o que ocorre apenas é uma organização grosseira do arranjo, com os valores menores tendendo a ir (dentro de sua própria subsequência) para o início do arranjo e os maiores ficando mais para o final.

Como esse processo é aplicado para vários incrementos diferentes, cada um menor que seu anterior, a cada etapa as subsequências ficam com mais elementos e as "distâncias" entre as comparações se reduz. Isso significa que o arranjo tende a ficar com uma ordenação melhor a cada passo. Adicionalmente, a cada novo incremento utilizado, cada subsequência ordenada já se aproveita das ordenações grosseiras dos passos anteriores, de forma que a estratégia do inserção direta se aproveita das reduções das comparações e deslocamentos de dados da repetição interna.

O último passo é a aplicação do inserção direta, o que equivale efetivamente a usar incremento igual a 1. Quando isso acontece, o arranjo já tem sua ordenação geral bastante melhorada e o melhor caso da ordenação é válido para praticamente todo o arranjo.

Pelo uso dessa estratégia, a ordenação passa a ter desempenho substancialmente superior a qualquer um dos métodos básicos apresentados na seção anterior.

Uma questão em aberto no algoritmo é a determinação de quais são os incrementos que devem ser usados. De forma geral, testes empíricos sugerem que bons desempenhos podem ser obtidos seguindo-se algumas sugestões[7] simples:

1. O último passo usa incremento 1 (o que equivale à aplicação do inserção direta).
2. Cada incremento anterior pode ser calculado como o incremento do próximo passo multiplicado por 3 e acrescido de 1. Assim, se um incremento for p, então o incremento anterior deve ser $3p + 1$.
3. O número total de incrementos diferentes a ser usados deve ser equivalente a $\log_3 n$.

Exemplificando: para um vetor com 100.000 elementos devem ser usados 10 incrementos diferentes ($\log_3$ 100.000 vale aproximadamente 10,48). Cada incremento fica definido como: 29524, 9841, 3280, 1093, 364, 121, 40, 13, 4 e 1 (calculados do último para o primeiro).

A análise da complexidade do *shellsort* não é trivial e muito do seu desempenho foi medido por análises experimentais. De forma geral, funções na forma $T(\boldsymbol{n}) = n\log^2 n$ ou mesmo $T(n) = n\log n$ representam satisfatoriamente seu comportamento. Em geral, o *shellsort* é considerado um algoritmo $O(n\log^2 n)$.

7 Há algumas variações destas sugestões, dependendo dos diversos autores. Portanto, elas não podem ser consideradas regras.

Algoritmo 3-15

```
1  shellsort(dados[]: inteiro, tamanho:inteiro)
2     { define os incrementos }
3     númeroIncrementos ← log(tamanho)/log(3)
4     p[0] = 1 { último incremento }
5     para i ← 1 até númeroIncrementos - 1 faça
6           p[i] ← p[i – 1] * 3 + 1
7     fim-para
8
9     { ordenação das subsequências para cada incremento }
10    i ← númeroIncrementos – 1  { começa pelo maior }
11    enquanto i > 0 faça
12          { ordena cada subsequência }
13          incremento ← p[i]
14          para j ← incremento até tamanho – 1 faça
15                auxiliar ← dados[j]
16
17                { deslocamentos dentro da mesma subsequência }
18                k ← j – incremento
19                enquanto dados[k] > auxiliar e k ≥ 0 faça
20                      dados[k + incremento] ← dados[k]
21                      k ← k - incremento
22                fim-enquanto
23                dados[k + incremento] ← auxiliar
24          fim-para
25
26          { passa para o próximo incremento }
27          i ← i - 1
28    fim-enquanto
```

O Algoritmo 3-15 mostra uma implementação para o *shellsort*, que é um "algoritmo de inserção" (que insere um valor em relação aos demais já ordenados). Para entender a parte de ordenação das subsequências é interessante notar que elas não são tratadas separadamente. Na realidade, a repetição da linha 2.2.2 passa por todos os elementos do arranjo, mas o uso da variável *incremento* (nas linhas 2.2.2 a 2.2.2) garantem que cada comparação seja restrita a sua própria subsequência.

Outro exemplo de "algoritmo de seleção", ou seja, daquele que seleciona qual item deve ocupar cada posição, é o *heapsort*. A cada passo, um elemento é selecionado para ocupar cada posição do arranjo.

Para tornar a escolha do item selecionada a cada passo mais eficiente, os dados são inicialmente arrumados para permitirem uma busca binária. Essa organização recebe o nome de *monte*, que segue a estrutura de uma árvore binária.

Essa solução é uma alternativa bastante prática, já que a árvore é organizada no próprio arranjo de dados, sem a necessidade de qualquer estrutura adicional. O arranjo pode ser visto como uma árvore binária se, considerando que a posição *i* do arranjo representa um nó, as posições 2*i* + 1 e 2*i* + 2 forem usadas para armazenar os filhos esquerdo e direito. A raiz é escolhida como a posição zero, ou seja, o primeiro elemento do vetor. A determinação de qual nó é pai ou filho de outro se torna simples e dispensa os usuais ponteiros.

Tome-se como exemplo o arranjo de inteiros da Figura 5 A visão usual dos dados do arranjo é como uma lista linear de dados. Porém, considerando-se a posição zero a raiz, as posições 1 e 2 são seus filhos esquerdo e direito, quando a visão é alterada para uma árvore. Dessa forma, a posição 3 do arranjo tem seu filho esquerdo na posição 7 e o filho direito na posição 8. A árvore em si não existe, mas o arranjo é usado como se fosse uma árvore, apenas pela manipulação adequada dos índices das posições.

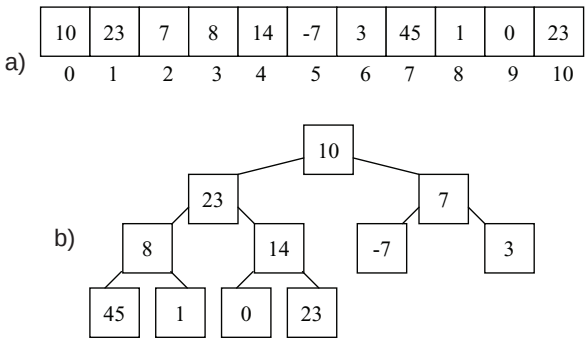


Figura 5 Ilustração de um arranjo de números inteiros com 11 posições. (a) Visão usual dos dados no arranjo; (b) Visão do arranjo como uma árvore binária.

O monte[8] é uma árvore binária com a seguinte propriedade: o valor de um dado nó é sempre maior ou igual ao de seus dois filhos. Dessa propriedade decorre que a raiz armazena o maior elemento da árvore.

Para a ordenação pelo *heapsort*, então, é preciso inicialmente criar o monte com os dados do arranjo. Assim os elementos são trocados de posição até que os elementos do arranjo, sob a visão de árvore binária, formem um monte.

O algoritmo para realizar essa tarefa é apresentado no Algoritmo 3-16. Para cada elemento do arranjo, do segundo até o último, é feita sua "subida" na árvore em direção à raiz, desde que seu valor seja maior que o de seu pai. Assim, pela estrutura da árvore, os valores maiores migram, um a um, até sua posição correta na árvore. O resultado é que o maior valor fica armazenado na raiz. A complexidade desta fase é O(*n*log *n*): o comando *para* é linear, enquanto a repetição interna do *enquanto* é O(log *n*), pois usa a estrutura binária da árvore.

Algoritmo 3-16

```
1     { criação do monte }
2     para i ← 1 até tamanho – 1 faça
3           auxiliar ← dados[i]
4           filho ← i
5           pai ← (filho – 1)/2  { posição do pai }
6           enquanto filho > 0 e dados[pai] < auxiliar faça
7                 dados[filho] ← dados[pai]
8                 filho ← pai
9                 pai ← (filho – 1)/2
10          fim-enquanto
11          dados[pai] ← auxiliar
12    fim-para
```

Criado o monte, pode ser feita então a ordenação por seleção. Para essa segunda fase, é feita a seleção do maior elemento do arranjo (que está na raiz) e este é trocado com o que está na última posição do arranjo. Com a troca, o monte é deixado incorreto, pois na raiz há um elemento qualquer. O ajuste do monte é feito fazendo com que o item que acabou de ser colocado na raiz "desça" na

8 No original, monte é chamado *heap*.

estrutura da árvore até sua posição. O elemento ordenado é considerado fora do monte.

O processo é repetido para o segundo maior elemento, que agora está na raiz da árvore, fazendo-se a troca para a posição correta e novo ajuste do monte. O processo é repetido até a raiz.

O monte é usado para que a seleção dos elementos seja eficiente, permitindo que, um a um, cada valor seja colocado na posição correta, do maior até o menor.

Algoritmo 3-17

```
1   { ordenação com uso do monte }
2   i ← tamanho – 1
3   enquanto i < 2 faça
4     { coloca o valor da raiz na posição definitiva }
5     auxiliar ← dados[0] { raiz }
6     dados[i] ← dados[0]
7     dados[0] ← auxiliar
8
9     { corrige o monte }
10    auxiliar ← dados[0]  { valor incorreto, colocado na raiz }
11    pai ← 0
12    se dados[2*pai + 1] > dados[2*pai + 2] e 2*pai + 2 ≥ i então
13          filho ← 2*pai + 1
14    senão
15          filho ← 2*pai + 2
16    fim-se
17    enquanto pai < i e dados[filho] < dados[pai] faça
18          dados[pai] ← dados[filho]
19          pai ← filho
20          se dados[2*pai + 1] > dados[2*pai + 2] e 2*pai + 2 ≥ i então
21                filho ← 2*pai + 1
```

```
22          senão
23                filho ← 2*pai + 2
24          fim-se
25    fim-enquanto
26    dados[filho] ← auxiliar
27   fim-enquanto
```

No Algoritmo 3-17 é apresentada a parte da ordenação do *heapsort*. A troca da raiz (maior elemento atualmente no monte) coloca o valor na posição correta, pois o maior vai para a última posição, o segundo maior para a penúltima posição e assim por diante. O reajuste do monte é feito fazendo com que o valor colocado na raiz desça pela árvore até sua posição correta. Para isso é escolhido o filho que possui o maior valor, o que determina qual ramo da árvore será usado para a descida. A descida pela árvore é proporcional a log *n* e o processo é repetido para cada elemento do arranjo. A complexidade da fase de ordenação é, portanto, O(*n*log *n*).

Em termos gerais, o *heapsort* possui duas fases O(*n*log *n*) consecutivas e é, portanto, O(*n*log *n*) como um todo.

O algoritmo do *heapsort* não faz qualquer consideração sobre como os dados estão no arranjo. Isso significa que, mesmo que os dados já estejam ordenados, haverá a troca de suas posições para que o monte seja criado e depois a ordenação seja feita. O custo total do tempo de execução é distribuído praticamente metade para cada fase, o que o torna computacionalmente pesado para arranjos pequenos, mas interessante para arranjos com grande quantidade de itens para ser ordenados.

É importante salientar que, usualmente, a implementação do *heapsort* se apresenta em algumas formas diferentes. Este é o caso da implementação indicada por Drozdek (2002) [Erro: Origem da referência não encontrada]. Os princípios discutidos aqui, entretanto, estão presentes.

Existe, ainda, um último algoritmo para ordenação, o qual é amplamente utilizado. Ele é chamado de *quicksort*, que quer dizer método rápido de ordenação. Essencialmente recursivo em sua implementação, utiliza o algoritmo de particionamento para efetuar a ordenação.

Algoritmo 3-18

```
1  partição(dados[]:inteiro,início, fim: inteiro, var esquerda, direita: inteiro)
2  { particionamento dos dados entre as posições início e fim do arranjo }
3      esquerda ← início
4      direita ← fim
5      pivô ← dados[(início + fim)/2] { elemento do meio }
6      enquanto esquerda ≤ direita faça
7            { procura um valor maior ou igual ao pivô }
8            enquanto dados[esquerda] < pivô faça
9                  esquerda ← esquerda + 1
10           fim-enquanto
11
12           { procura um valor menor ou igual ao pivô }
13           enquanto dados[direita] > pivô faça
14                 direita ← direita - 1
15           fim-enquanto
16
17           { faz a troca }
18           se esquerda ≤ direita então
19                 auxiliar ← dados[esquerda]
20                 dados[esquerda] ← dados[direita]
21                 dados[direita] ← auxiliar
22                 direita ← direita - 1
23                 esquerda ← esquerda + 1
24           fim-se
25     fim-enquanto
```

O Algoritmo 3-18 mostra o particionamento. Para um dado intervalo dentro do arranjo, delimitado pelos índices *início* e *fim*, são feitas as trocas necessárias. Um valor de referência, chamado *pivô*, é escolhido e as trocas seguem um princípio simples: todos os valores menores que *pivô* devem ficar à esquerda (isto

é, no início do intervalo) e os maiores à direita. Os índices *esquerda* e *direita* são usados para buscar os valores e as trocas são feitas conforme necessário. Deve-se notar que este procedimento não se preocupa com qualquer aspecto da ordenação, pois apenas separa os dados em dois conjuntos: menores ou iguais ao pivô de um lado, maiores ou iguais ao pivô de outro[9]. A propriedade mais interessante desse particionamento é que, depois de concluído, cada partição se torna independente. Em outras palavras, os elementos que ficaram em uma partição devem apenas ser ordenados naquela partição e nunca ocorrem trocas entre elementos de partições diferentes.

O *quicksort* aplica o particionamento descrito para o arranjo como um todo e depois para cada partição criada. As partições ficam, então, cada vez menores e quando não atingem tamanho menor que 2, significa que estão ordenadas.

Algoritmo 3-19

```
1   quicksort(dados[]: inteiro, tamanho: inteiro)
2     quickRecursivo(dados, 0, tamanho – 1)
3
4
5   quickRecursivo(dados[]: inteiro, início, fim: inteiro)
6     partição(dados, início, fim, esquerda, direita)
7
8     { ordena primeira partição }
9     se início < direita então
10          quickRecursivo(dados, início, direita)
11    fim-se
12
13    { ordena segunda partição }
14    se esquerda < fim então
15          quickRecursivo(dados, esquerda, fim)
16    fim-se
```

9 Os valores que forem iguais ao do pivô podem ficar tanto na partição da esquerda quanto da direita, sendo indiferente para o funcionamento do algoritmo.

No Algoritmo 3-19, *quicksort* é usado para chamar a parte recursiva do algoritmo, definindo o arranjo inteiro como a partição inicial. Para cada chamada de *quickRecursivo*, a partição é subdividida em duas novas partições e cada uma delas, se tiver mais que um elemento, é repassada para nova subdivisão. A primeira partição é definida de *início* até *direita* e a segunda de *esquerda* até *fim*.

Em termos de desempenho, o *quicksort* tem seu melhor caso quando o pivô escolhido para a partição leva à criação de duas novas partições de tamanhos equivalentes (idealmente dividindo a partição anterior em duas de mesmo comprimento). Neste caso, o algoritmo tem desempenho O(*n*log *n*). Entretanto as partições podem ter tamanhos muito diferentes, o que é prejudicial. No pior caso, a escolha do pivô faz com que uma partição de *k* elementos seja subdividida em uma partição de tamanho 1 e outra de *k* – 1. Nesta situação crítica e isso ocorrendo para todos os particionamentos, o *quicksort* cai para desempenho O(n^2). O fato é que, para arranjos usuais, o pior caso é uma hipótese remota e o desempenho O(*n*log *n*) é a situação comum como caso médio.

Comparativamente aos outros métodos de ordenação mais avançados (*shellsort* e *heapsort*), o *quicksort* é muito leve e apresenta, para arranjos com muitos elementos, um desempenho geralmente muito superior aos demais.

2.3 Ordenação na memória secundária

Diferentemente dos métodos usados em memória principal, a ordenação em memória secundária tem, necessariamente, que se preocupar com os acessos a disco. Como o acesso a dados em disco é feito por blocos, leituras e gravações desnecessárias se tornam uma penalização excessiva para o tempo de execução.

O princípio das técnicas usadas para ordenação em disco é tentar fazer acesso sequencial aos dados, o que minimiza acesso a registros isolados. Assim, todos os registros lidos em um bloco são processados.

Porém, inicialmente, será apresentada a solução padrão de um método de ordenação por fusão, conhecido como *mergesort*. O método se baseia na "fusão" de listas ordenadas, ou seja, na criação de uma lista ordenada a partir de duas outras listas também já ordenadas.

A criação de uma lista ordenada a partir de outras duas também ordenadas é simples. É sempre necessário ver o início de cada lista, sendo que o menor deles é movido para lista de saída. Esse processo é repetido até que uma das duas listas se esgote e então, o restante dos itens da lista remanescente é movido para a lista de saída. Este processo é exemplificado, para memória

principal, para o caso da geração de um arranjo, chamado *listaSaída*, a partir de outros dois, *lista1* e *lista2* (Algoritmo 3-20).

Algoritmo 3-20

```
1     { fusão de duas listas ordenadas
2     i ← 0
3     i1 ← 0
4     i2 ← 0
5     enquanto i1 < tamanhoLista1 e i2 < tamanhoLista2 faça  { até
      o fim de uma das listas }
6           se lista1[i1] < lista2[i2] então
7                 listaSaída[i] ← lista1[i1]
8                 i1 ← i1 + 1
9           senão
10                listaSaída[i] ← lista2[i2]
11                i2 ← i2 + 1
12          fim-se
13          i ← i + 1
14    fim-enquanto
15
16    enquanto i1 < tamanhoLista1 faça { restante da lista 1, se houver }
17          listaSaída[i] ← lista1[i1]
18          i ← i + 1
19          i1 ← i1 + 1
20    fim-enquanto
21
22    enquanto i2 < tamanhoLista2 faça {restante da lista 2, se houver}
23          listaSaída[i] ← lista2[i2]
24          i ← i + 1
25          i2 ← i2 + 1
26    fim-enquanto
```

Para usar esse processo na ordenação, parte-se do princípio que uma lista com um único elemento se encontra sempre ordenada. Então, duas listas, de um elemento cada uma, são ordenadas em outra lista com dois elementos; depois, listas com 2 elementos são ordenadas em listas com 4 itens, e assim sucessivamente até que se obtenha, por fusões sucessivas, a ordenação de todo o conjunto.

Um detalhe importante é que a fusão de listas requer espaço disponível para armazenar a lista de saída. Uma observação atenta do código do Algoritmo 3-20 permite verificar que as variáveis *lista1* e *lista2* não podem ser reaproveitadas para guardar a saída, de forma que outra variável, *listaSaída*, é necessária para armazenar o resultado da intercalação.

O *mergesort* faz uma abordagem recursiva para a ordenação, dividindo o arranjo a ser ordenado pela metade passo a passo, até que cada parte tenha comprimento igual a 1 (ou seja, um único elemento). A partir daí, volta fazendo as fusões sucessivas, até obter a ordenação completa.

Algoritmo 3-21

```
1   mergesort(dados[]: inteiro, tamanho: inteiro)
2     mergeRecursivo(dados, 0, tamanho – 1)  { chama recursiva-
mente para todos os dados }
3
4
5   mergeRecursivo(dados[]: inteiro, início, fim: inteiro)
6     { ordena recursivamente duas partições }
7     se início < fim então
8           meio ← (início + fim)/2
9           mergeRecursivo(dados, início, meio) {primeira metade}
10          mergeRecursivo(dados, meio + 1, fim)   {segunda metade}
11
12          { fusão das metades ordenadas }
13          fusão(dados, início, fim)
14    fim-se
```

O *mergesort* é apresentado no Algoritmo 3-21, no qual a primeira parte faz apenas a chamada para o procedimento recursivo. O procedimento *mergeRecursivo* faz o trabalho, dividindo todos as partições do arranjo (se forem maior que 1) em duas e acionando o procedimento para cada uma delas.

Ao terminar a execução das duas chamadas *mergeRecursivo* das linhas 2.3 e 2.3, as duas partições estarão ordenadas. Então, o procedimento de fusão (linha 2.3) é chamado para fazer a intercalação dos dois segmentos do arranjo. Na prática, o procedimento *fusão* é uma adaptação da fusão do Algoritmo 3-20. Em *fusão*, os equivalentes a *lista1* e *lista2* são partes do arranjo, definidas de *início* até *início* + *meio* e de *meio* + *1* até *fim*, sendo *meio* igual a *(início* + *fim)/2*. Um arranjo temporário é necessário para guardar o resultado da fusão, a qual, quando terminada, é copiada de volta para o arranjo.

Em termos de desempenho, o tempo de execução do *mergesort* é O(*n*log *n*), o que o equipara aos algoritmos avançados já descritos. O detalhe é que, necessariamente, um arranjo auxiliar tem que ser alocado para guardar os valores das intercalações, exigindo o dobro do espaço originalmente usado para guardar o vetor de dados.

A utilização do *mergesort* para arquivos utiliza algumas modificações importantes. Em primeiro lugar, as listas a serem mescladas estão em disco, assim como em disco deve ser armazenado o resultado da fusão. Realizar a fusão de apenas duas listas também é contraproducente, dado que o número de vezes que cada item deve lido para a memória aumenta. Finalmente, não é de se esperar que o início seja a fusão de uma grande quantidade de arquivos com apenas um registro.

Uma solução para a ordenação de dados em arquivo, portanto, pode ser estabelecida pelos seguintes passos:

1. Carregar tantos registros do arquivo quanto caibam na memória principal, ordenando-os com um método apropriado para memória principal (*quicksort*, por exemplo) e gravando-os como uma sequência ordenada em um arquivo separado.
2. Realizar a fusão de *k* arquivos com listas ordenadas, gerando uma nova lista ordenada, a qual é também armazenada em disco. Repetir o processo de fusão com as listas existentes até que uma única lista seja criada.

Supondo que o arquivo seja bastante grande e que não caibam todos os dados em memória, o passo 1 acima viabiliza a criação de listas ordenadas com um conjunto razoável de registros. Como o custo para recuperar os dados do disco e regravá-los é mais significativo que a ordenação em memória principal,

o custo da ordenação inicial de cada lista não é tão relevante no processo como um todo. Esta primeira fase é completada fazendo-se apenas uma leitura e uma gravação de cada registro em disco.

A fusão de *k* arquivos é feita adaptando-se a fusão de listas para *k* listas, e não apenas para 2, como apresentado no *mergesort* original. O valor de *k* depende de quantos blocos (um de cada arquivo a ser intercalado) podem ser mantidos simultaneamente em memória principal. Para cada fusão, os dados são lidos apenas uma vez do disco e gravados também apenas uma vez. Se o valor de *k* permitir que todos os arquivos do primeiro passo sejam fundidos de uma vez (ou seja, se o número de sequências geradas for menor ou igual a *k*), então cada registro será lido e escrito apenas mais uma vez.

2.4 Considerações finais

A necessidade de ordenação é de incontestável importância e a escolha de um método adequado para cada situação é essencial. Para tanto, foram apresentados os principais métodos elementares de ordenação, cujos tempos são $O(n^2)$, englobando o *inserção direta*, o *seleção direta* e o *bubblesort*. Para estes métodos, o desempenho para quantidades pequenas de dados é razoável, o que permite fazer a escolha pelos dois primeiros em algumas situações. O *bubblesort*, pelo grande número de trocas, não deve ser usado.

Para os métodos avançados em memória principal, sua aplicação é para grandes volumes de dados que possam ser simultaneamente guardados em um vetor. A opção pelo *shellsort*, *heapsort* e *quicksort* é, em geral, indiferente. Porém, o *quicksort* é, de longe, o mais popular, com um desempenho excepcional. O *shellsort* necessita de um arranjo auxiliar para guardar os incrementos e o *heapsort* não considera pré-ordenações existentes.

Finalmente, quando a situação recai para a ordenação de dados em arquivos, mudam-se todos os critérios a serem considerados. Como a leitura e escrita de dados passam a ser o cerne do consumo de tempo, quanto menos os dados forem lidos e gravados, melhor o desempenho final. O uso de alternativas mistas como a descrita, fazendo ordenações em memória principal do que for possível e usando o *mergesort* adaptado para realizar as fusões necessárias, caracteriza uma estratégia importante para esta situação.

Não existem métodos absolutamente piores ou melhores. A situação da aplicação é que determina a escolha por um ou outro método de ordenação.

UNIDADE 3

Representação da informação

3.1 Primeiras palavras

Para que se possa ter domínio sobre como as informações são manipuladas por computadores, é preciso, inicialmente, ter uma boa noção de como elas são representadas.

A representação de informações está, em um primeiro momento, diretamente relacionada a códigos. Por exemplo, convencionamos que o símbolo **A** representa o fonema que todos conhecem (existente, por exemplo, no início da palavra *abacate*). Para que os códigos funcionem, é preciso que todos que compartilhem tal código conheçam sua representação. Cada um consegue fazer a leitura deste texto porque conhece o código de representação das letras, o código de combinação das letras para a formação de palavras, o código usado para dar significado a cada palavra e os códigos de sintaxe e semântica que permitem dar significado a este conjunto de letras, espaços e símbolos de pontuação.

Para que este conceito fique mais claro, um exemplo de representação numérica pode ser utilizado, como a clássica numeração em números romanos. O código romano que indica que o valor 1 é representado por I, o valor 5 por V, 10 por X, 50 por L, 100 por C, 500 por D e 1000 por M. Na formação de outros valores, combinações precisas são usadas: III para o 3, IV para o 4, IX para o 9, etc., o que é estabelecido por regras de soma (XI é X somado a I, ou seja, 11) e subtração (XL é X subtraído de L, ou seja, 40). Sabendo-se o código (símbolos e regras usadas para a interpretação), fica "fácil" traduzir MCMLXXI para 1981. (Fica aqui um desafio ao leitor para dizer como os romanos indicavam o número zero.)

Um segundo exemplo relativo a valores numéricos é o próprio sistema decimal, amplamente utilizado por todos. Os símbolos incluem os dígitos de 0 a 9 e as regras de representação assumem grandezas diferentes dependendo da posição do dígito do número. Assim, o dígito 8 em 18 significa 8 unidades, enquanto o mesmo dígito em 81 significa 8 dezenas. O uso de uma vírgula para partes decimais também pode ser empregado, atribuindo potências negativas de 10 ao significado do dígito, de forma que o dígito 8 em 1,48 passa a indicar 8 centésimos, por exemplo. E isso sem considerar números positivos ou negativos.

Fugindo dos exemplos numéricos, existe a representação alfabética, usada para a composição de palavras. As letras de A a Z são usadas para formar palavras e a justaposição delas determina como elas devem ser lidas. Além destes, outros símbolos como acentos e cedilhas podem ser usados para modificar o significado das palavras (como análise e analise). Um dos problemas da representação da fala (oral) por palavras (escrita) é que o som associado às combinações de letras não é preciso. Os casos mais clássicos são as dificuldades com G e J, que por terem um som parecido, faz com que muitas pessoas se

confundam com a grafia correta (berinjela ou beringela?), o mesmo ocorrendo com o X (chícara ou xícara?). Ou, ainda, quantas pessoas sabem que a pronúncia correta de *sintaxe* é "sintas*s*e" e não "sinta*c*se"? O problema de codificações "abertas", ou seja, em que pode haver várias representações para uma única informação, é a imprecisão na interpretação, o que não pode ser tolerado no armazenamento de dados.

Outras formas de codificação da informação são os números em base 2, os números em hexadecimal, os alfabetos grego e cirílico, as placas de trânsito, o código Morse e a escrita em *Braile*, por exemplo.

Em um segundo momento, para que uma informação possa ser representada, é preciso salientar que existem alguns elementos importantes:

1. Um conjunto de símbolos deve ser escolhido.
2. Regras para combinar os símbolos devem ser estabelecidas.
3. Os agentes que vão representar ou interpretar estes símbolos devem conhecer e concordar com a codificação.

Limitar o conjunto de símbolos e conhecê-lo por completo é importante para a interpretação. As regras definem como as associações de símbolos devem ser interpretadas e em quais contextos se aplicam. Finalmente, quem vai armazenar e quem vai recuperar as informações devem ser conhecedores dos símbolos, regras e contexto, para que a informação que é armazenada seja a mesma que é recuperada.

3.2 Representação básica da informação

Nos computadores, há restrições severas sobre o que usar para representar as informações usando códigos. Assim, como os computadores apenas possuem *bits*, estes passam a ser a única forma de representar as informações. É sabido, também, que cada bit representa apenas 0 ou 1 (desligado ou ligado, apagado ou aceso, ausente ou presente, etc.). Portanto, restam, para qualquer representação, apenas os códigos binários, ou seja, formados por zeros e uns.

Cabe, agora, definir como cada código é associado a um significado.

Uma primeira regra, já de longa data, é agrupar os bits em conjuntos de 8, chamando a isso *byte*. Como cada byte tem 8 bits e cada bit assume apenas 2 valores, são possíveis exatas 256 combinações para cada byte, indo de 00000000 a 11111111.

A representação de informação usando os bytes fica, assim, condicionada a:

1. Definir os 256 possíveis bytes como o conjunto de símbolos que podem ser usados.
2. Utilizar conjuntos de um ou mais bytes para combinar os símbolos básicos e definir novos símbolos.
3. Definir, em comum acordo com o agende que vai guardar a informação e o agente que vai interpretá-la, o que cada combinação de símbolos significa.

Alguns poucos exemplos serão dados, a seguir, para mostrar como representar alguns tipos de informação.

Números inteiros sem sinal

Uma das formas de representar um número inteiro positivo usando os bytes como símbolos é fazer a associação direta entre a representação binária do número e os bits dos bytes. Desta forma, o número 33, que tem representação binária 100001_2, é representado pelo byte 00100001. Da mesma forma, 66 é associado ao byte 01000010, enquanto 255 ao byte 11111111.

De posse da regra e concordando com a representação, se houver um byte com o valor 10001010 e ele representar um valor inteiro positivo, pode-se saber, com segurança, que o valor representado é 138.

A Figura 6 mostra um exemplo de quatro bytes, cujos valores representam, segundo a codificação convencionada, os valores inteiros 8, 1, 127 e 10, respectivamente. Neste exemplo, são quatro valores distintos, cada um deles formado por apenas um byte e usando a convenção da representação binária sem sinal para associar os valores a cada byte em particular.

10001000	00000001	10000001	00001010

Figura 6 Quatro bytes, cada um representando um valor inteiro sem sinal distinto. Na ordem: 136, 1, 129 e 10.

Com essa representação usando apenas um byte, é possível apenas representar os números inteiros de 0 a 255. Ampliar estes valores segue a mesma regra usada nos números decimais: se valores maiores precisam de mais dígitos, valores maiores precisam de mais bytes.

Outra representação para números inteiros sem sinal, bastante comum em computação, usa conjuntos de quatro bytes. Alinhados, os quatro bytes representam 32 bits e permitem 2^{32} combinações, ou seja, valores de 0 a 4.294.967.295. Deve-se notar que a ordem em que os bytes são considerados também é importante e esta escolha também faz parte da regra de codificação. Usando quatro bytes para representar um único valor, o 1.140.899.978, os bytes ficariam como os representados na Figura 7.

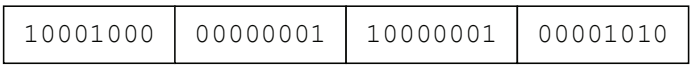


Figura 7 Quatro bytes que, juntos, representam um único valor inteiro sem sinal, de 32 bits. O valor binário é o $10001000000000110000001000010 10_2$, ou seja, 1.140.899.978 em decimal.

Em C ou C++, estes dois tipos de representação de números inteiros apresentados corresponderiam ao *unsigned char* (quando usado como inteiro) e ao *unsigned int*, respectivamente.

Um ponto importante a ser observado é que as representações têm restrições. O número inteiro 300 não pode ser representado segundo as regras definidas para uso de apenas um byte. Mas essas limitações estão presentes em várias outras codificações, como a inexistência de casas decimais em números romanos.

Números inteiros com sinal

A representação de números negativos em bytes utiliza, em sua forma mais comum, o primeiro bit para indicar o sinal. Se seu valor for 0, o número é positivo; se for 1, negativo. Porém, como um dos bits é usado para o sinal, resta um bit a menos para representar o valor em si.

Usando um único byte, portanto, o primeiro bit indica o sinal e os sete bits restantes são usados para os números. Sendo o primeiro bit igual a zero, os valores representados vão de 0 (00000000) a 127 (01111111). As outras 127 combinações com o primeiro bit igual a 1 representam os valores de -1 a -128.

Se forem usados quatro bytes para armazenar um valor inteiro com sinal, então, os valores possíveis se iniciam em –2.147.483.648 e vão até 2.147.483.647, dada a reserva do primeiro bit para o sinal.

Quatro bytes, cada um representando um valor inteiro com sinal, são ilustrados na Figura 8. O primeiro e o terceiro (da esquerda para a direita) são negativos, com o primeiro bit igual a 1; o segundo e o quarto são positivos.

10001000	00000001	10000001	00001010

Figura 8 Representação de quatro inteiros com sinal, cada um com um byte. Valores representados, na ordem: -120, 1, -127 e 10.

A Figura 9, por sua vez, mostra um único valor de quatro bytes, que é negativo por seu primeiro bit (neste caso o do primeiro byte) ser igual a 1.

10001000	00000001	10000001	00001010

Figura 9 Quatro bytes que, juntos, representam um único valor inteiro com sinal, de 32 bits. O valor, em decimal, é -2.013.167.350.

Uma observação cabe aqui: a interpretação de números negativos não é direta nesses exemplos, pois foi usada a notação chamada *complemento para 2*. Isso significa que o byte 10000000 representa -128 e não "menos zero". Essa representação não será detalhada neste texto.

Em C ou C++, esses tipos corresponderiam ao *char* (quando usado como inteiro) e ao *int*, respectivamente.

Números reais

Um tipo de dado importante em computação é o que representa os números chamados "números com ponto flutuante", que são os valores com parte decimal.

Valores reais podem ser representados por uma convenção não tão óbvia, mas padronizada para vários sistemas. Esse é o caso da especificação IEEE 754 [İ], que utiliza subconjuntos dos 32 bits para a representação.

Um valor real é codificado, usando essa especificação, dividindo-o em mantissa (m) e expoente (e). Um valor qualquer é representado no formato $m2^e$. A parte referente à mantissa utiliza 23 bits e o expoente usa 8 bits. Um bit para representação de sinal completa os 32 bits.

Numerando cada bit, da esquerda para a direita, de 0 a 31, então o bit 0 indica o sinal da mantissa, os bits de 1 a 9 representam o expoente e os demais, de 10 a 31 formam a mantissa. A Figura 10 mostra a combinação de bits, usando quatro bytes e a convenção da especificação IEEE, que representa o valor 100,5.

01000010	11001001	00000000	00000000

Figura 10 Um número em ponto flutuante usando quatro bytes consecutivos. O valor representado é 100,5.

Em C ou C++, esse exemplo corresponde às variáveis do tipo *float*.

Caracteres

O uso dos bytes na representação de caracteres é feito por uma associação arbitrária entre os caracteres e as combinações de bits.

Em algum momento, convencionou-se que o desenho *b* (ou *b*) fosse usado para representar a letra “bê”, sendo que qualquer outro símbolo poderia ter sido escolhido. Os gregos, por exemplo, optaram por .

A convenção mais comum de representação de caracteres em sistemas computacionais é dada pela chamada tabela ASCII (American Standard Code for Information Interchange, ou código padrão americano para intercâmbio de informações). Essa tabela define, arbitrariamente, a qual código de bits está associado cada caractere. Por exemplo, o dígito *0* é representado pelo byte 00110000, o *J* pelo 01001010, o *j* pelo 01101010 e o **%** pelo 00100101.

Além da tabela ASCII, outras tabelas também populares incluem a EBCDIC (Extended Binary Coded Decimal Interchange Code, ou código de intercâmbio estendido de decimais codificados em binário) e a UNICODE (Unicode Standard, ou padrão de “código único”). A codificação UTF-16, em particular, é uma versão da representação UNICODE de 16 bits (2 bytes) para representar caracteres.

Cadeias de caracteres

Cadeias de caracteres, usadas para a representação de textos, são usualmente sequências de bytes que representam caracteres. Um controle adicional deve ser acrescido à sequência para controlar quantos caracteres são válidos.

Aqui são comentadas duas formas comuns para a representação de cadeias de caracteres: a da linguagem C padrão e a da linguagem Pascal.

Em C, as cadeias de caracteres são representadas como arranjos (vetores) de caracteres. São considerados válidos, para o texto armazenado, todos os caracteres a partir do início do arranjo até que seja encontrado um byte com valor 00000000, indicado na linguagem por ‘\0’. Na linguagem Pascal, opta-se por reservar o primeiro byte do tipo *string* para armazenar o número de caracteres

que devem ser considerados como válidos. Como apenas um byte é usado para indicar essa quantidade, há um máximo de 255 caracteres para esse tipo de dados.

A Figura 11 mostra uma representação de uma variável para guardar textos em C. Declarada como uma cadeia de caracteres de 6 bytes, esta representa os caracteres *AABC* (usando codificação ASCII). O quinto byte indica o fim da sequência válida; os caracteres seguintes são desprezados.

01000001	01000001	01000010	01000011	00000000	01000001

Figura 11 Representação de quatro bytes para uma variável declarada como *char[6]* e armazenando o literal *AABC*.

3.3 Informações na memória principal

Na memória principal, as informações são armazenadas nos bytes disponíveis, usando representações como as descritas no tópico anterior, além de várias outras. A forma mais usual de definir uma área da memória para uso é por meio das variáveis dos programas; outra forma é por meio de alocação dinâmica de memória.

O tratamento de especificidades como tipos de memória, o que envolveria memórias RAM e *cache*, entre outras, não pertence ao contexto deste texto. Assim, salvo em ocasiões deixadas explícitas, assume-se que o termo memória apenas se refere à memória principal, ou seja, a uma memória RAM comum.

Como características da memória principal, dentro do escopo deste livro, destacam-se:

1. A área de memória é composta de bytes dispostos de forma sequencial, sem divisões.
2. Cada byte possui seu endereço de memória.
3. O tempo de acesso (consulta do conteúdo ou armazenamento de padrão de bits) a qualquer byte da memória é considerado similar, sendo feito muito rapidamente.
4. A memória é considerada volátil.
5. A quantidade total de bytes existentes para armazenamento (tamanho da memória) é considerada de tamanho arbitrário, porém finito.

Para o entendimento do uso da memória é preciso conhecer como os bytes são usados para armazenar as informações. Esse conceito passa pela representação da informação, que pode ocorrer conforme os exemplos indicados no tópico anterior.

Como ilustração do conceito, pode-se considerar a declaração de variáveis em um programa escrito na linguagem C, apresentada no Algoritmo 1-22.

Algoritmo 1-22

```
1     /* exemplos de declaracoes */
2     int i;
3     char n[5];
4     float f;
```

A variável **i** é um inteiro com sinal e ocupa quatro bytes consecutivos na memória; **n** consome outros cinco bytes consecutivos, sendo usada para armazenar um caractere por byte; a variável *f*, assim como *i*, ocupa quatro bytes consecutivos e serve para armazenar um valor em ponto flutuante.

É válido lembrar que não há diferença entre os quatro bytes da variável *i* dos da variável *f*. Como ocorre com qualquer byte na memória, não existem diferenças entre um e outro. Assim, reconhecer como os bytes da variável inteira são diferenciados da variável *float* é apenas uma convenção, ou seja, é preciso saber *o que* está armazenado. No programa, os bytes reservados para a variável *i* serão sempre interpretados como um valor inteiro com sinal e seguem, assim, a interpretação adequada para seus 32 bits. No caso da variável *f*, a interpretação como um valor IEEE 754 é definida pelo tipo da variável.

Um exemplo alternativo, sobre conhecer uma representação para interpretar corretamente um símbolo, corresponde ao símbolo *X*. Ao interpretar *X* como letra, trata-se do “xis”; mas ao defini-lo como um numeral romano, dá-se-lhe um valor, 10. Ter apenas o símbolo não é suficiente para entender que informação ele representa; é requerido que se tenha em mãos também a convenção que deve ser usada na interpretação.

Assim, não é padrão que haja, na memória, alguma referência ao tipo de seu conteúdo. A função de interpretar corretamente a codificação usada para representar a informação é de responsabilidade de quem usa o dado. É possível, por exemplo, que os bytes da variável inteira sejam interpretados como

um *float*[10]; nessa situação, os bits seriam interpretados de forma distinta e os valores reconhecidos seriam completamente diferentes.

Essa interpretação diferenciada do mesmo padrão de bits na memória pode ser observada desde a Figura 6 até a Figura 9, nas quais o mesmo padrão de bits é interpretado de diferentes formas , dependendo apenas das convenções usadas.

3.4 Informações na memória secundária

O termo memória secundária é aplicado às formas de armazenamento "fora" do computador. Em geral, a memória secundária se refere a um dispositivo de armazenamento em massa, como é o caso dos discos rígidos, memórias *flash* (*pen drives* e cartões de memória) e discos ópticos, entre outros.

Memórias secundárias se caracterizam com comportamento diferenciado da memória principal, justamente em função de suas características físicas e lógicas. As principais características da memória secundária incluem:

1. A área de memória é composta de bytes dispostos de forma sequencial, com divisões dependentes do dispositivo.
2. Cada byte ou conjunto de bytes possui seu endereço no dispositivo.
3. O tempo de acesso (consulta do conteúdo ou armazenamento de padrão de bits) a cada byte depende das características do dispositivo no qual é feito o armazenamento, sendo muito mais lento quando comparado ao da memória principal.
4. O armazenamento é permanente.
5. O tamanho da memória secundária é limitado e, em geral, superior à capacidade de armazenamento da memória principal.

Assim como na memória principal, uma codificação é escolhida para armazenar cada informação, utilizando as mesmas convenções empregadas na memória principal. Isso quer dizer que um valor inteiro com sinal de dois bytes é representado na memória secundária da mesma forma que na memória principal, sendo raramente necessárias conversões de representação.

A principal diferença entre a memória secundária e a primária é o tempo de acesso. Qualquer que seja a mídia usada para o armazenamento secundário, o tempo de acesso é muito superior, enquanto o tempo de acesso à memória é da ordem de nanossegundos (10^{-9}s), um disco rígido moderno consome tempo na

10 Isso é viável fazendo um ponteiro para *float* apontar para o endereço da variável inteira.

ordem de milissegundos (10^{-3}s); isto dá uma razão de um milhão de vezes. Outros dispositivos como CDs ou *pen drives*, embora apresentem razões de tempo diferentes das dos discos rígidos, também são significativamente diferentes da memória principal.

Um fator extremamente relevante quando se fala em memória principal e secundária é que o processador somente tem acesso aos dados da memória principal. Ao se necessitar de um dado armazenado em disco, por exemplo, é necessário primeiro transferi-lo para a memória principal. Nesta última o dado é trabalhado e, eventualmente, transferido de volta ao disco para armazenamento.

Como o armazenamento de dados está atualmente concentrado em discos rígidos, o enfoque neste livro será dado ao armazenamento secundário nesse tipo de dispositivo.

3.4.1 Discos rígidos como memória secundária

Os discos rígidos são a principal forma de armazenamento permanente de dados, estando presentes desde pequenos notebooks até grandes servidores.

Sendo o tempo de acesso ao disco muito maior que à memória principal e dada a necessidade de que o dado esteja presente nesta última para que seja manipulado pelo computador, o tempo de acesso é a característica mais importante que deve ser considerada quando se trabalha com armazenamento secundário.

Assim, é preciso que se tenha uma boa ideia de como funcionam os discos rígidos e as razões que influenciam o tempo para se armazenar ou recuperar um dado.

Essa discussão passa por entender a geometria básica dos discos rígidos, como o sistema operacional (ou um SGBD[11]) organiza os dados em disco e quais os principais retardos que influenciam no tempo de acesso.

Geometria dos discos rígidos

Os discos rígidos representam uma classe de armazenamento secundário que utilizam magnetismo. Vários discos metálicos giram em torno de um eixo e cabeças de leitura e gravação são usadas para obter ou guardar bits sobre a superfície destes discos (Figura 12).

11 Sistema gerenciador de banco de dados.

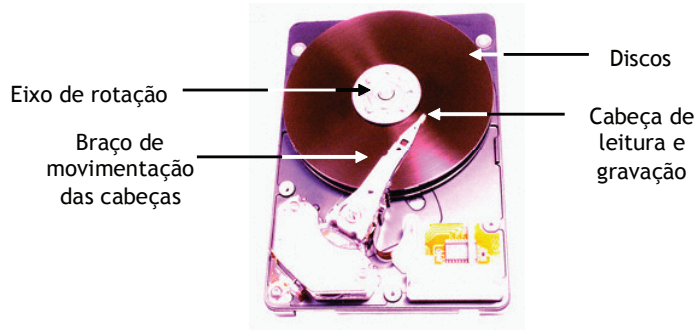


Figura 12 Disco rígido.

Fonte: Imagem de Corbys Holding, Inc. (http://www.corbisimages.com/)

Os bits são gravados ao longo de círculos concêntricos (trilhas), sendo que cada círculo é usualmente segmentado em trechos regulares (setores). A Figura 13 mostra uma distribuição de 9 trilhas, divididas em 8 setores, sobre uma das superfícies do disco. Os bits são gravados ao longo de cada trilha, distribuídos em seus setores.

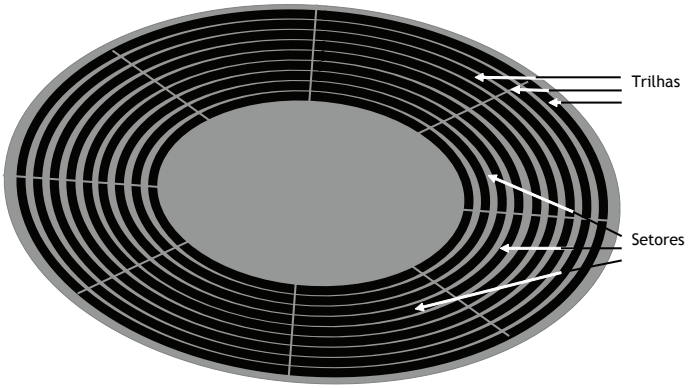


Figura 13 Ilustração da superfície de um dos discos, indicando as trilhas (círculos concêntricos) e os setores (segmentos das trilhas).

Um disco rígido, na prática, é constituído de vários discos que giram juntos. Cada disco permite a gravação em ambas as faces, e cada uma das faces tem sua própria cabeça de leitura e gravação. As cabeças são presas a um braço (Figura 13), de modo que todas se movimentam como um único bloco. Enquanto o braço faz com que as cabeças se movimentem radialmente, o disco gira sob elas. As gravações e leituras ocorrem enquanto o setor de uma dada trilha está sob a cabeça de gravação.

A capacidade de armazenamento de um disco é em função do número de superfícies, do número de trilhas, do número de setores e da capacidade, em bytes, de armazenamento de cada setor. Usualmente um setor contém uma quantidade de bytes dada como uma potência de 2. Um dos discos rígidos da Seagate,

de 1 terabyte contém, por exemplo, um total de 1.953.525.168 setores de 512 bytes cada um (Seagate Technology).

O sistema operacional e a recuperação de dados

O sistema operacional de um computador é o responsável por se organizar para localizar os dados no disco rígido. Isso significa definir a forma de organizar a área de dados para boot, fazer o gerenciamento das listas com os nomes e atributos dos arquivos e estruturar a localização de tais arquivos no disco rígido. Estabelece-se, assim, um mapeamento entre as trilhas e setores existentes no disco rígido e a organização definida pelo sistema operacional.

A forma como cada sistema operacional ou SGBD define sua organização de arquivos e dados não faz parte dos interesses deste texto. Assim, fica sugerido ao leitor que procure informações adicionais pesquisando por sistemas de arquivos como EXT3, ReiserFS, FAT32 e NTFS, entre outros.

Dentro do escopo deste livro, apenas algumas características dessa organização são consideradas relevantes. Embora muitas outras características sejam consideradas importantes, não serão tratadas aqui por motivos de simplificação.

Os sistemas operacionais, em função do sistema de arquivos escolhido, definem um tamanho padrão para transferências de dados do disco rígido ou para ele. Esse tamanho representa um *bloco* de disco, também conhecido por *cluster* ou *página* de disco.

Um bloco, usualmente dado por uma quantidade de bytes que seja potência de 2, permite ao sistema reservar espaços nas memórias *cache* e principal para agilizar as transferências de dados e dividir o espaço em disco para acomodar os diversos arquivos.

Para este texto, um bloco é considerado como uma quantidade fixa[12] de bytes que podem ser armazenada ou recuperada do disco rígido de cada vez. Em outras palavras, o sistema é capaz de transferir somente esta quantidade de bytes de cada vez; nem mais, nem menos. Assume-se, também, que o espaço em memória (denominado *buffer*) para realizar essas transferências seja equivalente a exatamente um bloco.

Supondo, a título de exemplo, que o sistema operacional utilize blocos de 2 KiB (2048 bytes) para seu sistema de arquivos e que já se saiba em quais setores do disco um determinado arquivo se encontra, a leitura dos dados pode ser realizada. Considerando que os dados correspondam a um texto simples

12 Alguns sistemas de arquivos permitem blocos de tamanhos variados. Estes não serão considerados nesta disciplina.

que necessite 1578 bytes (entre caracteres, espaços, mudanças de linhas, tabulações e etc.), a leitura dos dados para a memória deve ser solicitada por um programa. O sistema operacional repassa ao *hardware* as instruções para a leitura, o que resulta na transferência de exatos 2048 bytes (um bloco) para o *buffer* na memória principal. Terminada essa transferência, os 1578 primeiros bytes do *buffer* são repassados para o programa. O sistema se encarrega do controle para que os bytes restantes no *buffer* não sejam considerados como dados válidos.

Sob as mesmas condições, um segundo exemplo é um programa solicitando a leitura apenas da primeira linha do mesmo arquivo texto, assumindo que tal linha tenha 56 bytes. A solicitação de leitura é passada para o sistema operacional, que executa a transferência de um bloco do disco para o *buffer*. A partir do *buffer*, os 56 primeiros bytes são passados ao programa. Se, posteriormente o programa solicitar outra linha (supondo agora outros 75 bytes), os próximos 75 bytes do *buffer* são repassados deste para o programa, não necessitando novo acesso a disco, visto que os dados já se encontram na memória. Novas solicitações do programa por dados irão consumindo os bytes do *buffer*, até completar os 1578 bytes válidos ali armazenados.

Havendo um arquivo que ocupe mais do que um bloco em disco rígido, uma nova leitura a disco ocorrerá somente quando os bytes do primeiro bloco já tiverem sido consumidos. Assim que o *buffer* é liberado, uma nova leitura é feita (isto é, um novo acesso a disco) e os dados continuam a ser repassados para o programa. Qualquer arquivo maior que 2048 bytes, dadas as condições do exemplo descrito anteriormente, está sujeito a essa situação.

O processo de gravação é similar. Os programas transferem dados, por meio de comandos de escrita, para o *buffer* do sistema operacional. O sistema decide quando esses dados são efetivamente transferidos para o disco. De qualquer forma, cada gravação de dados em disco rígido exige a transferência de um bloco completo.

Naturalmente as descrições acima são simplificações da realidade. Podem existir vários *buffers* (de um bloco cada um), tanto para leitura quanto para escrita, pode haver memórias *cache* tanto no circuito do disco rígido quanto na placa mãe e, além disso, o sistema não cuida apenas de um único programa e um único arquivo, mas de vários, simultaneamente. Para justificar as organizações de dados da parte de arquivos, entretanto, esta simplificação apresentada é bastante útil.

Adicionalmente, além do sistema operacional, alguns SGBDs podem controlar diretamente uma partição do disco rígido e manter sua própria estrutura de mapeamento de dados. Nesses casos, o SGBD retira do sistema operacional

esse controle. Os conceitos de blocos e de sua utilização, porém, permanecem inalterados.

Tempos de acesso para armazenamento e recuperação

Mesmo com o progresso que experimenta a tecnologia, o acesso a um disco rígido é muito limitado por suas características mecânicas. O envolvimento de movimentos de partes mecânicas, como o braço das cabeças de leitura e gravação, acrescenta latências importantes: como quebrar a inércia do braço parado, acelerá-lo e movimentá-lo em direção à trilha correta, desacelerá-lo até parar sobre a trilha desejada para, finalmente, aguardar a informação passar sob a cabeça e realizar a transferência de dados.

Dos diversos fatores que influenciam o tempo e geram *atrasos*, ou *latências*, são elencados três:

1. Latência de busca;
2. Latência de rotação;
3. Latência de transferência.

Todo dado que precisa ser transferido do disco, ou para ele, depende da cabeça de leitura e gravação estar posicionada sobre o dado. Isto requer, em um primeiro momento, mover o braço de forma a posicionar as cabeças sobre a trilha correta. O tempo despedido para fazer essa movimentação gera a *latência de busca*. Quanto mais longe da trilha correta estiver o braço, maior o tempo necessário para acertar a posição. Em sistemas multitarefas, as várias requisições simultâneas de acesso são ordenadas para minimizar a movimentação do braço e, assim, reduzir o tempo de seu deslocamento. Portanto, não importa a ordem em que as solicitações de acesso chegam, pois o atendimento a cada uma delas segue a menor necessidade de movimentação do braço.

A *latência de rotação* é o tempo necessário para que, estando a cabeça de leitura e gravação sobre a trilha correta, o início dos dados passe sob ela. Em algumas situações os bytes poderão estar quase que imediatamente disponíveis ou, no pior caso, uma rotação completa tem que ser aguardada.

Estando no posicionamento correto, é preciso agora transferir os dados, ou seja, gravar o bloco no disco ou recuperá-lo. O tempo gasto na transferência em si é chamado *latência de transferência* e depende do número de bytes transferidos, da quantidade de bytes contidos na trilha e da velocidade de rotação do disco.

3.5 Considerações finais

Nesta primeira unidade foram vistos vários conceitos importantes:

1. Representação da informação.
2. Características da memória principal.
3. Características da memória secundária e acesso a discos rígidos.

Uma questão essencial é lembrar que, para representar uma informação útil, é preciso um código. De posse do código, é possível avaliar um conjunto de símbolos e interpretá-lo para obter a informação representada. Nos computadores, os bits representam os dados básicos e são organizados em bytes. Sabendo-se o que um dado conjunto de bytes armazena, torna-se viável analisar o padrão de bits que ele possui e interpretar a informação nele contida.

A memória principal é uma coleção finita de bytes e é usada para armazenar as diversas informações. Tal tipo de armazenamento é rápido, porém bastante limitado em tamanho, e o custo relativo desse tipo de memória é um fator restritivo para seu tamanho.

Memória secundária é um termo que se associa a formas de armazenamento de grande capacidade, porém com acesso muito mais lento que aquele à memória principal. Discos rígidos são exemplos de dispositivos de armazenamento secundário, cujas características mecânicas retardam o acesso a um dado conjunto de bytes.

O sistema operacional ou um SGBD são os responsáveis por organizar os dados dos discos rígidos e definem, para se ter acesso, uma estruturação baseada em blocos. Sendo os blocos as unidades de transferência de dados, entender como todo o processo de gravação ou leitura de dados ocorre permite projetar estratégias inteligentes para se obter desempenho de acesso à memória secundária.

UNIDADE 4

Organização de dados em memória principal

4.1 Primeiras palavras

A memória principal é o elemento fundamental de armazenamento de informações, sem o qual o conceito de computação não existiria como é hoje. A organização de dados em memória principal cobre desde variáveis simples, como as inteiras e reais, até estruturas mais sofisticadas, como árvores com nós alocados dinamicamente. Em um ponto intermediário nessa organização tão vasta, estão os arranjos e matrizes, que são estruturas naturalmente projetadas para que façam o armazenamento organizado de um conjunto relativamente grande de dados.

O contexto desta unidade envolve algumas formas importantes para armazenamento de dados usando vetores. Em particular, assume-se aqui que os dados sejam registros com informações variadas (sobre alunos, com RA, nome e curso, por exemplo, ou então sobre produtos, com código, descrição e valor). Desse ponto de vista, as discussões serão em torno de estruturas de dados contendo vetores de registros, chamados aqui de *tabelas*.

Este texto cobre, inicialmente, como são as estruturas gerais e as operações usuais sobre elas. A partir dessas estruturações, são abordadas e discutidas formas de organizar os dados, de modo que as operações sobre as tabelas sejam coerentes. Ao longo das discussões, as vantagens e desvantagens, bem como a aplicabilidade das organizações apresentadas, serão abordadas.

4.2 Tabelas em memória principal e operações sobre elas

Tabelas em memória principal, segundo o contexto desta unidade, são apenas arranjos de registros. Assim, assume-se simplesmente que os dados sejam declarados como vetores. O Algoritmo 4-23 mostra um exemplo de como podem ser as declarações dos tipos envolvidos, enquanto a Figura 14 ilustra como uma variável declarada como *tTabelaAlunos* pode ser visualizada.

Algoritmo 4-23

```
1   { exemplo de declaração }
2   tipo tAluno: registro
3            ra: inteiro
4            nome: literal
5            códigoCurso: inteiro
6            ira: inteiro
```

```
7      fim-registro
8
9   constante tamanho: inteiro = 10000
10  tipo tTabelaAlunos: tAluno[tamanho]
```

	ra	nome	códigoCurso	ira
0	417763	Romualdo José	05	15822
1	407262	Antonio Albuquerque	02	16883
2	447611	Solvino da Silva	02	14223
3	478827	Dora Constância	04	15309
4	412098	Fernanda Camargo	13	12401
	...	...	...	...
tamanho - 1				

Figura 14 Exemplo de uma tabela contendo dados de acordo com as declarações do Algoritmo 4-23. Cada linha representa um registro, sendo indicada pelo número da posição; os campos estão dispostos como colunas, com os identificadores mostrados anteriormente.

Nas discussões feitas neste texto, para que não haja a necessidade de contextualizar exemplos diferentes para cada caso, a tabela será genérica (isto é, não associada a um caso particular) e os conteúdos mínimos serão apresentados nas figuras. Em especial, o termo *chave* será utilizado para indicar um campo qualquer importante em uma situação. O leitor deve ter em conta, portanto, que a chave pode ser, em um momento, o nome do aluno, em outro, seu RA, e assim por diante.

Para que um vetor possa ser usado, é preciso indicar um conjunto de operações que podem ser feitas sobre ele. Assim, para cada organização serão tratadas as operações:

Criação (ou população);

1. Inserção;
2. Remoção;
3. Pesquisa (ou busca);
4. Atualização;
5. Manutenção.

A *criação* de uma tabela corresponde ao seu preenchimento inicial com os dados disponíveis. Assume-se, inicialmente, que o vetor não contém dados

e que deve ser preenchido com um conjunto pré-existente de registros[13]. Para efetuar essa operação é preciso conhecer a organização que será utilizada para manter a coerência com as demais operações. Como o resultado da operação é o preenchimento da tabela, termos como *popular a tabela* ou *operação de população* são comuns.

A operação de *inserção* equivale ao acréscimo de um novo registro a uma tabela já existente. Em termos estruturais, deve manter a tabela organizada e ser coerente com as demais operações. A tabela existente pode ter qualquer número de registros (incluindo estar vazia) e requisitos de espaço disponível entram em consideração.

A *remoção* de um registro da tabela equivale a, dada a posição na tabela, a qual contém o registro que não mais deve fazer parte do conjunto, excluí-lo dos registros válidos. A não ser que esteja explicitamente escrito, a operação de remoção já assume que houve uma pesquisa anterior para localizar o item a ser removido, de forma que a localização não é considerada como parte da operação em si.

Localizar um item na tabela é a principal razão para o armazenamento de dados. A operação de *pesquisa* envolve um ou mais algoritmos de localização de um registro. Para isso, define-se uma *chave de pesquisa*, para a qual se deve localizar um registro que a contenha na tabela. Em geral, assume-se que não haja repetições da chave, de forma que qualquer pesquisa identifica apenas um dos registros do conjunto[14].

Comum como operação, a *atualização* corresponde à modificação de um ou mais dos campos de um dado registro. Um dado pode estar simplesmente errado (o nome foi digitado erroneamente) ou necessitar ser atualizado (o IRA foi recalculado e precisa ser modificado). Assim como na remoção, a atualização considera a localização como uma operação à parte. Modificar um registro total ou parcialmente pode influir nas demais operações e, assim, algoritmos específicos podem ser necessários para manter a organização geral da tabela.

Finalmente, algumas opções para a implementação das operações sobre as tabelas acabam gerando situações em que pesquisas ou novas inserções, por exemplo, acabam perdendo desempenho. Uma *manutenção* em uma tabela é uma operação que reorganiza os dados, eliminando possíveis ocorrências de problemas. Como um exemplo, a remoção pode simplesmente ser feita marcando um registro como inválido, usando para isso um campo lógico associado a

13 Esses dados podem estar em um arquivo ou precisam ser digitados, por exemplo.

14 Esta consideração de um único registro para cada chave é bastante restritiva. O caso de vários registros com a mesma chave é comum e também importante. Como exemplo, a busca na tabela da Figura 4-14. usando como chave o código do curso, que retorna, para um único código, vários registros.

ele; uma manutenção faria a efetiva remoção dos registros marcados como inválidos. Esse tipo de operação, quando necessária, pode ser feita sob demanda ou periodicamente.

4.3 Formas de organização

Partindo da forma mais simples para a mais complexa, serão vistas como tabelas podem ser mantidas sob três pontos de vista diferentes. As tabelas podem ser mantidas *sem ordenação*; *ordenadas* por algum critério ou, então, usar uma organização de acesso direto, conhecida como *tabela hash*.

Cada uma dessas formas será tratada a seguir.

4.3.1 Tabelas sem ordenação

Uma tabela sem ordenação é simplesmente uma coleção de registros em um vetor de dados, sem que se determine uma relação de qual registro vem antes ou depois de outro. As tabelas sem ordenação são tratadas sob dois pontos de vista diferentes, para cada uma das operações estabelecidas. Ao final é apresentada uma discussão de quando podem ou não ser utilizadas.

Os dois enfoques compreendem:

1. Disposição compacta dos registros.
2. Disposição distribuída dos registros.

Disposição compacta

A manutenção de dados em um vetor, em sua forma usual, consiste em alocar uma quantidade de posições máxima (chamada neste texto de *tamanho*) e utilizar as posições iniciais para guardar os dados existentes. Essa forma de organização, na qual os registros válidos estão sempre nas primeiras posições do vetor, corresponde à disposição compacta. Para determinar quais são os registros válidos, um controle adicional é necessário. Esse controle é feito, na maioria das vezes, simplesmente armazenando a quantidade de registros.

Deste modo, um vetor de dados com **n** registros armazenados, terá suas posições de 0 a **n** – 1 com registros válidos, com as demais posições sendo consideradas disponíveis para armazenamento e cujo conteúdo é irrelevante e, portanto, desconsiderado.

A *criação* de uma tabela não ordenada a partir de um conjunto de dados inicial é trivial. Basta que os dados sejam colocados sequencialmente no vetor, iniciando-se na posição 0. Colocados todos os dados, o controle do número de itens **– n –** deve conter a quantidade correta de dados. O Algoritmo 4-24 ilustra como o conjunto de dados, considerado aqui em um arquivo, são colocados na tabela. Essa operação, considerando que o tamanho do problema seja o número inicial de registros no arquivo, consome tempo de execução O(*n*). Caso não haja dados iniciais, basta que **n** seja mantido igual a zero, o que indica que a tabela está vazia.

Algoritmo 4-24

```
1     { criação }
2     n ← 0
3     enquanto não fda(arquivoDados) e n < tamanho faça
4           leiaArquivo(arquivoDados, dado)
5
6           tabela[n] ← registro
7           n ← n + 1
8     fim-enquanto
```

Uma vez que a tabela esteja inicialmente estruturada com **n** registros (com **n** variando de zero a *tamanho*), é possível começar a utilizá-la.

O acréscimo de um novo registro à tabela, dado que não há ordem relativa entre os dados, pode ser feito simplesmente acrescentando o novo item à próxima posição disponível. O Algoritmo 4-25 mostra a operação de *inserção*, cujo tempo independe de qualquer fator, sendo sempre constante. Assim esta operação é O(1).

Algoritmo 4-25

```
1     insira(dado)
2           se n < tamanho então
3                 tabela[n] ← dado
4                 n ← n + 1
5           fim-se
```

A *remoção* de um registro pode ser feita de forma que o item indicado apenas deixe de fazer parte do conjunto. Assim, dada a posição do registro, indicada por *p* no Algoritmo 4-26, sua eliminação pode ser feita apenas com a cópia do registro da última posição sobre a posição do registro removido e, então, reduzindo-se a quantidade de itens válidos.

Algoritmo 4-26

```
1     remova(p)
2           tabela[p] ← tabela[n – 1]
3           n ← n - 1
```

Para a remoção, assume-se apenas que *p* contenha um valor válido, no intervalo entre 0 e **n**. Com tempo constante para execução, a remoção pertence a O(1).

A *pesquisa* por um dado registro exige uma busca sequencial, um a um, consultando cada registro. A busca sequencial é apresentada no Algoritmo 4-27, no qual o parâmetro denominado *chave* substitui um dos campos do registro.

Algoritmo 4-27

```
1  busca(chave): inteiro
2     i ← 0
3     enquanto tabela[i].chave ≠ chave e i < n faça
4           i ← i + 1
5     fim-enquanto
6     se i < n então
7           retorne i  { posição onde achou o registro }
8     senão
9           retorne -1 { indicação de que não achou }
10    fim-se
```

No Algoritmo 4-27, o campo *chave* normalmente não possui esse nome. Na realidade, o campo especificado é o utilizado, realmente, na pesquisa, como o caso de nome, RA, código de produto, etc.

A pesquisa sequencial pode ser considerada sob duas situações: pesquisas com sucesso e sem sucesso. Uma *pesquisa com sucesso* é a aquela em que a chave usada na busca se encontra no vetor; a *sem sucesso* é quando o registro buscado não está presente. Na pesquisa com sucesso, se cada chave presente tiver a mesma probabilidade de ser utilizada em uma pesquisa, então se espera uma média de *n*/2 comparações até que se encontre um registro qualquer, sendo *n* o número de registros presente no vetor. Isso faz com que a pesquisa, no caso médio, seja O(*n*). Quando não há sucesso na pesquisa, somente após realizar comparações com todos os registros é que se chega à conclusão de que o registro não se encontra na tabela. Nesta última situação, considerada o pior caso, são sempre feitas *n* comparações e o tempo de execução também é O(*n*).

Uma *alteração* feita em um dado qualquer de uma tabela sem ordenação não representa problema, pois nenhuma de suas funcionalidades e operações fica prejudicada. Assim, as alterações podem ser feitas sem requerer qualquer ajuste adicional.

Situação similar ocorre para o caso da *manutenção*. Cada operação deixa a tabela em sua forma mais adequada, de forma que não há qualquer necessidade periódica de reajustes. Não existe, assim, manutenção nessa forma de organização.

Disposição distribuída

Outra forma de organizar uma tabela sem ordenação é não requerer que os dados fiquem nas primeiras posições do vetor. Nessa alternativa, cada registro tem associado a ele um campo lógico, o qual indica se o registro armazenado na posição é ou não válido. Por exemplo, pode-se usar o valor *verdadeiro* para indicar que o registro é válido e *falso* caso não o seja.

A *criação* da tabela não ordenada a partir de um conjunto de dados inicial é simples. Como na solução anterior, basta que os dados sejam colocados sequencialmente no vetor, iniciando-se na posição 0. Para cada posição, o valor *verdadeiro* deve ser colocado no campo de validade. Para todas as demais posições do arranjo, o valor falso deve, necessariamente, ser colocado na validade. O Algoritmo 4-28 mostra essa construção.

Algoritmo 4-28

```
1     { criação }
2     i ← 0
3     enquanto não fda(arquivoDados) e i < tamanho faça
4           leiaArquivo(arquivoDados, dado)
5
6           tabela[i] ← registro
7           tabela[i].validade ← verdadeiro
8           i ← i + 1
9     fim-enquanto
10
11    para i ← i até tamanho – 1 faça
12          tabela[i].validade ← falso
13    fim-para
```

A avaliação do tempo necessário para criar uma tabela desse tipo deve ser feita com cuidado. Deve-se notar, inicialmente, que todo o arranjo é modificado, destacando-se a atribuição de *verdadeiro* ou *falso* ao campo *validade* de cada posição do vetor. Porém, o maior custo para esse código é o da atribuição feita no primeiro laço de repetição, no qual os registros são copiados para as posições do arranjo. Nesse caso, quanto mais registros, maior o custo desta execução. Assim, é razoável considerar que o tamanho do problema é o número de novos registros que serão inseridos e que o tempo é proporcional a ele de forma linear. O tempo de execução pode ser considerado O(*n*).

A *inserção* de um novo registro à tabela exige que ele seja colocado em uma posição vazia, o que implica em uma busca para que este espaço livre seja determinado. Essa busca é necessária, pois registros removidos (como será apresentado mais adiante) são marcados com *validade* igual a *falso*. O Algoritmo 4-29 mostra esta operação. O tempo de execução desse código está relacionado ao número de posições livres e com o número de registros inseridos. Como a busca pela posição vazia sempre se inicia na posição zero, a tendência é que, após um tempo, as posições vazias tendam a ficar sempre para o final do arranjo. Assim, quanto mais dados inseridos, maior a demora para localizar uma posição livre e maior o tempo de inserção. Remoções quebram essa regra, pois podem deixar uma posição disponível logo no início do arranjo.

Algoritmo 4-29

```
1     insira(dado)
2           i ← 0
3           enquanto tabela[i].validade e i < tamanho faça
4                 i ← i + 1
5           fim-se
6           se i < tamanho então
7                 tabela[i] ← dado
8                 tabela[i].validade ← verdadeiro
9           fim-se
```

Para a *remoção* de um registro em uma posição *p*, somente é necessário que seu campo de validade seja marcado como *falso*. O Algoritmo 4-30 apresenta esta solução, que tem tempo O(1).

Algoritmo 4-30

```
1     remova(p)
2           tabela[p].validade ← falso
```

A *pesquisa* por um dado registro exige uma busca sequencial, um a um, consultando cada registro. A busca sequencial foi apresentada no Algoritmo 4-27, mas tem que ser modificada, pois não há controle de quantos registros existem na tabela. Assim, todas as buscas terminam segundo uma das seguintes condições: o registro foi encontrado ou todo o arranjo foi verificado. O tempo de execução para a pesquisa leva, em caso de sucesso e caso haja probabilidades iguais de qualquer chave ser consultada, tempo proporcional a metade do tamanho total do arranjo. Em caso de insucesso na pesquisa, todo o arranjo é verificado. Tecnicamente, o tempo é constante para ambos os casos, e o algoritmo é O(1). Porém, é importante notar que este O(1) é pior que qualquer O(*n*) usado na organização anterior, pois necessariamente o valor de *n* é inferior ao tamanho do vetor de dados.

O Algoritmo 4-31 apresenta a versão modificada da busca sequencial.

Algoritmo 4-31

```
1  busca(chave): inteiro
2   i ← 0
3   enquanto tabela[i].chave ≠ chave e tabela[i].validade e i < tamanho faça
4            i ← i + 1
5   fim-enquanto
6   se i < tamanho então
7            retorne i  { posição onde achou o registro }
8   senão
9            retorne -1 { indicação de que não achou }
10   fim-se
```

Como na situação da estruturação anterior, a *alteração* feita em um dado qualquer de uma tabela sem ordenação não representa problema, e pode ser feita sem requerer qualquer ajuste adicional.

Com o tempo, a tabela tende a ter seus dados distribuídos ao longo de todo o espaço disponível, com posições livres também dispersas ao longo da tabela. Para o caso da busca com sucesso, a situação em que os dados estejam concentrados nas primeiras posições favorece o algoritmo. Assim, uma operação de manutenção pode ser feita de tempos em tempos para que todos os registros marcados como válidos ocupem as primeiras posições da tabela. O algoritmo para realizar essa tarefa fica proposto como recomendação para o leitor.

A primeira alternativa de organização é superior à segunda, embora cada uma tenha seus méritos. A discussão seguinte considera apenas a primeira solução, quando são citadas as ordens de complexidade para as diversas operações.

De modo geral, dados sem ordenação sempre pecam no que se refere à pesquisa, que no melhor caso é O(*n*), com *n* sendo o número de itens existentes na tabela. Como abordado na próxima seção, a busca em tabelas ordenadas pode ser superior, porém inserções e remoções exigem maior esforço computacional.

Em termos de aplicabilidade, tabelas sem ordenação são importantes para pequenos conjuntos de dados, pois as inserções e remoções podem ser O(1) e a pesquisa O(*n*). Caso *n* seja relativamente pequeno, seu desempenho pode ser mais do que adequado para muitas aplicações. Entretanto, se o tamanho do problema aumentar, a pesquisa pode ser um elemento comprometedor, caso se opte por esta solução.

4.3.2 Tabelas ordenadas

Uma tabela, para ser considerada ordenada, depende de se estabelecer um critério de ordem entre seus registros. Em primeiro lugar, é preciso definir uma *chave de ordenação*, a qual, usualmente, é um dos campos do registro[15]. Dessa maneira, uma tabela de alunos pode ser ordenada por RA. Um segundo ponto é definir o critério para a ordenação, que pode ser o naturalmente usado nas relações entre valores numéricos (na qual o sentido de "menor" e "maior" são bastante conhecidos) ou ainda a ordem alfabética usada para os valores literais (com critérios similares, com "a" < "c" e "t" < "r", por exemplo). No caso particular da ordem alfabética, é preciso definir de antemão se "A" = "a", "o" = "ô" ou se "ç" = "c", de forma que as comparações "Antônio" = "Antonio" e "Lucas" = "LUCAS" reflitam os resultados desejados. Em geral, as cadeias de caracteres são modificadas para que as comparações se tornem similares à ordem alfabética usual.

Há valores que não possuem ordem natural, como as coordenadas no plano por exemplo. Nessa situação, não há um critério para se determinar se (10, 7) vem antes ou depois de (8, 8). Muitas aplicações definem critérios específicos para esses casos, embora não sejam naturais.

As tabelas ordenadas possuem uma vantagem excepcional para pesquisas por registros quando comparadas às não ordenadas, pois podem usar algoritmos mais sofisticados. Em contraposição, há o custo em se manter a tabela sempre ordenada.

Em termos de organização, as tabelas ordenadas mantêm seus registros todos nas primeiras posições do arranjo. Assim, existindo n registros na tabela, necessariamente eles se encontrarão nas posições de 0 a $n - 1$. Além disso, para uma posição i qualquer, o critério de ordenação deve ser respeitado, ou seja, a ordem do registro da posição $i - 1$ deve ser anterior ao da posição i, e o da posição $i + 1$ posterior. Neste caso, está sendo excluída a possibilidade de registros com chaves de ordenação iguais.

A *criação* de uma tabela ordenada pode ser feita pela inserção de todos os elementos disponíveis, um a um, sequencialmente nas primeiras posições da tabela, iniciando-se na posição zero. Feita a inserção inicial, um método de ordenação deve ser aplicado para deixar os dados em ordem crescente. O bom senso sugere que a opção seja por um método $O(n \log n)$, como é o caso do *quicksort*. Assim, a criação da tabela leva um tempo $O(n)$ para a colocação dos

15 A chave de ordenação também pode ser a composição de um ou mais campos. Como exemplo, considerando que um registro tenha campos separados para *nome* e *sobrenome*, a ordenação pode ser definida para a concatenação *nome* + " "+ *sobrenome*.

dados em uma ordem qualquer, seguida de um tempo O(*n* log *n*) para a ordenação, resultando em um tempo proporcional a O(*n* log *n*).

O Algoritmo 4-32 exemplifica a criação de uma tabela ordenada a partir de registros armazenados em um arquivo.

Algoritmo 4-32

```
1     { criação de tabela ordenada }
2     n ← 0
3     enquanto não fda(arquivoDados) e n < tamanho faça
4           leiaArquivo(arquivoDados, dado)
5
6           tabela[n] ← dado
7           n ← n + 1
8     fim-enquanto
9
10    quicksort(tabela, n)  { o critério do quicksort é o mesmo
da ordenação da tabela }
```

Qualquer novo registro adicionado a uma tabela ordenada manter o critério de ordenação inalterado. Portanto, a inserção precisa deslocar todos os registros que possuam chave maior que a do novo registro uma posição em direção ao final do arranjo, “abrindo” espaço para o novo item.

Algoritmo 4-33

```
1  insira(dado)
2     se n < tamanho então
3           { deslocamento dos maiores }
4           i ← n – 1  { posição do último registro válido }
5           enquanto tabela[i].chave > dado.chave e i > 0 faça
6                 tabela[i + 1] ← tabela[i]
7                 i ← i – 1
8           fim-enquanto
```

```
9
10          tabela[i + 1] ← dado  { coloca novo registro }
11          n ← n + 1
12    fim-se
```

O Algoritmo 4-33 apresenta o algoritmo de inserção. Se se considerar que, em média, metade dos registros existentes tenha de ser deslocada a cada inserção, o tempo de execução de *insira* é proporcional a *n*/2, ou que faz com que o algoritmo pertença a O(*n*). O pior caso também é O(*n*) e ocorre quando é inserido um registros cuja chave seja menor que a de todos os existentes na tabela. Como um detalhe adicional, sugere-se que o leitor faça a comparação desse algoritmo de inserção com o método de ordenação *inserção direta*.

Dada uma posição *p* da tabela, seu conteúdo pode ser excluído conforme o código do Algoritmo 4-34, o qual faz o deslocamento de todos os registros posteriores para "fechar" a posição do item removido, mantendo intactos os critérios de ordenação. Assume-se que o valor de p esteja entre 0 e **n** - 1 (inclusive). Considerando-se que, em média, metade dos registros deva ser movida em uma *remoção* qualquer, o tempo da exclusão é O(*n*). Também no pior caso o tempo é O(*n*), quando a remoção é do item de menor chave e todos os registros precisam ser deslocados.

Algoritmo 4-34

```
1    remova(p)
2     enquanto p ≤ n – 1 faça
3           tabela[p] ← tabela[p + 1]
4           p ← p + 1
5     fim-enquanto
6     n ← n - 1
```

O ponto forte das tabelas ordenadas é a pesquisa. Como os dados estão dispostos com uma relação específica entre si, algoritmos mais sofisticados podem ser empregados para localizar um dado registro, dada sua chave.

Uma forma muito interessante de pesquisa é a chamada *pesquisa binária*. Seu princípio de operação é prático: a cada passo, reduza pela metade o conjunto de dados que devem ser considerados. Os passos principais são:

1. Considere a partição atual como todo o arranjo.
2. Para a partição, divida-a pela metade e determine em qual delas está a chave de busca, terminando quando achar o registro ou quando a partição chegar a tamanho 1.

Algoritmo 4-35

```
1  buscaBinária(chave): inteiro
2     { partição inicial: todo o arranjo }
3     inícioPartição ← 0
4     fimPartição ← n – 1
5
6     { quebras sucessivas da partição pela metade }
7     enquanto fimPartição – inícioPartição > 0 então
8           metade ← (inícioPartição + fimPartição)/2
9
10          { decisão da partição }
11          se tabela[metade].chave = chave então
12                { achou }
13                inícioPartição ← metade
14                fimPartição ← metade {força o término da repetição}
15          senão
16                se tabela[metade].chave > chave então
17                      { está na primeira partição }
18                      fimPartição ← metade – 1
19                senão
20                      { está na segunda partição }
21                      inícioPartição ← metade + 1
22                fim-se
23          fim-se
24    fim-enquanto
25
```

```
26    se tabela[metade].chave = chave então
27          retorne metade
28    senão
29          retorne -1  { indicando que não localizou }
30    fim-se
```

Para esse código de pesquisa binária, o tamanho do problema é o número de registros presentes no arranjo, o qual também pode ser visto como o tamanho da partição considerada. No algoritmo, esse tamanho de partição é dado pela expressão *fimPartição — inícioPartição* + 1, refletida na condição do comando de repetição da linha 30 (uma unidade menor, no caso). A cada passo, o tamanho da partição é reduzido pela metade[16]. Daí a conclusão de que o controle do laço é dividido por dois a cada passo e, assim, a repetição pode ser considerada O(log *n*).

Uma alternativa de pesquisa em tabelas ordenadas é a chamada *pesquisa por estimativa* ou *pesquisa por interpolação*, embora possa ser aplicada sob condições específicas. Na pesquisa binária, há um "chute" no meio do arranjo, definindo uma posição de sondagem. Esse "chute" pode ser melhorado em algumas circunstâncias, o que agiliza a pesquisa. A pesquisa por estimativa tenta fazer este "chute" melhor tentando estimar a posição do registro procurado, e não apenas tentando na metade, como faz a pesquisa binária.

A estimativa da posição assume uma regra de proporção entre as chaves. Por exemplo, se em uma tabela a chave da posição 0 contiver o valor 100 e a da posição 500 contiver o valor 200, é razoável pensar que a chave 150 esteja próxima à posição 250, da mesma forma que a chave 105 esteja bem próxima ao início do arranjo e a chave 190 mais próxima ao fim. Por uma regra de proporção, a expressão de "chute" de uma posição pode ser dada por: (*posiçãoFim – posiçãoInício*)(*chaveBusca – chaveInício*)/(*chaveFim - chaveInício*) + *posiçãoInício*, sendo *posiçãoInício* e *posiçãoFim* os índices do início e do fim da partição, *chaveInício* e *chaveFim* os valores das chaves armazenadas nas posições do início e do fim da partição e, finalmente, *chaveBusca* o valor da chave que está sendo procurada.

Em termos de algoritmo para a pesquisa em tabelas ordenadas, a estrutura da pesquisa por estimativa é similar à da pesquisa binária, exceto pelo "chute" ser calculado pela expressão, e não ser dado exatamente no meio da partição. A cada repetição, com a mudança dos limites da partição, a estimativa é refeita.

16 Não exatamente pela metade, mas praticamente isso.

Algoritmo 4-36

```
1  buscaEstimativa(chave): inteiro
2     { partição inicial: todo o arranjo }
3     inícioPartição ← 0
4     fimPartição ← n – 1
5
6     { quebras sucessivas da partição usando estimativa }
7     enquanto fimPartição – inícioPartição > 0 então
8           posiçãoEstimada ← (fimPartição – inícioPartição) *
      (chave – tabela[inícioPartição].chave)/
9                 (tabela[fimPartição].chave –
      tabela[inícioPartição].chave) + inícioPartição
10
11          { decisão da partição }
12          se tabela[posiçãoEstimada].chave = chave então
13                { achou }
14                inícioPartição ← posiçãoEstimada
15                fimPartição ← posiçãoEstimada { força o término
      da repetição }
16          senão
17                se tabela[posiçãoEstimada].chave > chave então
18                      { está na primeira partição }
19                      fimPartição ← posiçãoEstimada – 1
20                senão
21                      { está na segunda partição }
22                      inícioPartição ← posiçãoEstimada + 1
23                fim-se
24          fim-se
25    fim-enquanto
26
27    se tabela[posiçãoEstimada].chave = chave então
```

```
28          retorne metade
29    senão
30          retorne -1  { indicando que não localizou }
31    fim-se
```

O Algoritmo 4-36 possui um tempo de execução que depende dos dados existentes. Caso a estimativa seja bem feita, seu desempenho é muito bom, tendendo a O(log(log *n*)). Porém, caso a estimativa gere alguma distorção, a busca pode degenerar e o desempenho ficar próximo a O(*n*), ou seja, similar a uma busca sequencial. Assim, a pesquisa por estimativa pode ser usada com eficiência apenas quando a distribuição das chaves for uniforme, podendo ser aproximada por uma reta; se não for este o caso, sua eficiência é completamente comprometida.

É importante lembrar que a pesquisa por outra chave, que não a de ordenação, usa a tabela como se esta fosse não ordenada. Nesse caso a pesquisa deve ser sequencial.

Alterações dos dados de um dado registro da tabela podem ser observadas sob dois pontos de vista. Caso a informação modificada não seja a chave de ordenação, a *alteração* não tem maiores consequências. Porém, se o dado alterado for a chave de ordenação, é provável que haja a necessidade de ajuste da tabela, de forma que a ordenação possa ser mantida. O Algoritmo 4-37 apresenta a operação de alteração.

Algoritmo 4-37

```
1   altere(p, dado)
2     se tabela[p].chave ≠ dado.chave então { a nova chave é
      diferente da anterior }
3           se tabela[p].chave > dado.chave então
4                 { ajusta para o início do arranjo }
5                 p ← p - 1
6                 enquanto tabela[p].chave > dado.chave e p ≥ 0 faça
7                       tabela[p + 1] ← tabela[p]
8                       p ← p - 1
9                 fim-enquanto
```

```
10                tabela[p + 1] ← dado
11          senão
12                { ajusta para o final do arranjo }
13                p ← p + 1
14                enquanto tabela[p].chave < dado.chave e p < n faça
15                      tabela[p - 1] ← tabela[p]
16                      p ← p + 1
17                fim-enquanto
18                tabela[p - 1] ← dado
19          fim-se
20    fim-se
```

Nesse algoritmo, assume-se que o registro que substitui o dado de certa posição é passado como parâmetro (*dado*), embora a modificação possa ocorrer internamente ao procedimento, com os devidos ajustes. Considerando que, quer seja a movimentação necessária em direção ao início quanto ao fim do arranjo, metade dos registros tenha que ser movimentado em média, a complexidade da alteração é O(*n*).

As tabelas ordenadas, estruturadas como descrito neste texto, são sempre mantidas de modo que todos os critérios de ordenação valham durante todo o tempo. Portanto, operações de *manutenção* não são necessárias.

4.3.3 Tabelas hash

A estratégia das *tabelas hash* é baseada em uma função, chamada de *função de escrutínio*, ou simplesmente *função hash*. A função hash é aplicada sobre a chave, usada para pesquisa e dá como resultado um endereço correspondente a uma posição na tabela. Assim, o resultado de uma função hash está restrita ao intervalo iniciado em 0 e indo até *tamanho* – 1.

Um exemplo inicial permite entender melhor o conceito das tabelas hash. Primeiramente, consideremos que os registros a serem inseridos possuam as seguintes chaves numéricas: 17, 138, 173, 294, 306, 472, 540, 551 e 618. Para armazenar estes 9 registros será usado um arranjo com 11 posições. A função hash escolhida pode ser dada por $h(c) = \lceil (c + 25)/64 \rceil$, expressão na qual $\lceil b \rceil$ indica o maior valor inteiro, menor ou igual a *b*. A Tabela 4 mostra o resultado da função aplicado às chaves da lista.

Tabela 4 Aplicação da função h(c) = ⌈(c + 25)/64⌉ para um conjunto de chaves.

c	h(*c*)
17	0
138	2
173	3
294	4
306	5
472	7
540	8
551	9
618	10

A aplicação da função hash, na prática, mostra em que posição do arranjo cada registro deve ser armazenado. Como exemplos, o registro com chave 138 é colocado na posição 2, enquanto o que possui a chave 540 fica na posição 8, notando que as posições 1 e 6 acabam não sendo usadas.

Para a pesquisa, basta também aplicar a função. Por exemplo, para localizar o registro com chave 472, calcula-se h(472), o que resulta em 7, que é onde o registro está efetivamente armazenado.

A organização de tabelas hash engloba uma estratégia que permite a recuperação muito rápida de um registro a partir de uma chave de pesquisa. Em situações ideais, apenas um cálculo pode ser necessário para localizar qualquer registro.

A grande dificuldade, entretanto, reside na escolha da função hash. Nem sempre é possível escolher uma função para a qual cada chave diferente produza um endereço diferente na tabela e, muitas vezes, é preciso lidar com registros diferentes que acabam mapeados para uma única posição. Pode-se notar que a função apresentada como exemplo se comporta bem apenas para o conjunto de dados existente. Qualquer chave diferente compromete todo o esquema.

Quando chaves distintas são mapeadas para uma mesma posição do arranjo, diz-se que houve uma *colisão*. Como colisões acabam sendo uma situação bastante comum, o tratamento destas deve fazer parte da estruturação de praticamente qualquer tabela hash.

Para abordar os diversos assuntos das tabelas hash, o texto faz, inicialmente, considerações sobre as funções hash, seguindo então para o tratamento de colisões e termina com considerações sobre as operações usuais sobre tabelas.

Uma função hash tem, inicialmente, que limitar que os valores calculados para endereços fiquem dentro do escopo da tabela, ou seja, no intervalo 0 a *tamanho* – 1. Outra característica importante é que sejam evitadas colisões, o que quer dizer que endereços repetidos devem ser evitados.

O projeto de uma função hash passa necessariamente pelo conhecimento das características das chaves que serão inseridas. Não é prático projetar a função sem entender como as chaves se comportam e quais seus valores prováveis.

Caso se tenha conhecimento de todas as chaves que serão inseridas, é possível projetar uma função *perfeita*, para a qual não há colisões. Situações dessa natureza ocorrem quando os dados são estáticos, como em uma tabela de palavras reservadas (de uma linguagem de programação) ou dicionários e listas telefônicas quando armazenados em mídia permanente. Nesses casos os dados não mudam e, quando mudarem, outra função deve ser elaborada.

No caso usual, somente são conhecidas as características gerais das chaves. Por exemplo, para uma tabela de alunos, com RA e nome, usando o campo RA para fazer o hash, sabe-se que a chave é numérica e possui duas partes: o número sequencial de matrícula e um dígito verificador (488732, por exemplo, é 48873 com dígito de controle 2). Para uma função hash sobre o RA, pode-se pensar em desprezar ou não o dígito de controle do cálculo. Se a tabela possuir 1000 posições, podem ser usados os dígitos da centena para posicionar o aluno. Como exemplo, 48873-2 seria posicionado no endereço 873. Considerando que a tabela armazene apenas alunos de uma turma, torna-se pouco provável que haja muitas colisões, pois somente as chaves que possuíssem dígitos de centena 873 seriam mapeadas para o endereço 873 (e.g., 468734, 478730, 498731 ou 508737), mas estes provavelmente seriam de outra turma, já que a numeração é sequencial.

Conhecer os dados ou pelo menos suas características ajudam a projetar uma função hash adequada (DROZDEK, 2002).

Há algumas técnicas usuais e simples que permitem definir funções com resultados interessantes.

Um primeiro ponto é que, como já apresentado, o valor resultante da função deve ser um endereço válido da tabela. Assim, pode-se usar a *divisão modular* para fazer essa restrição. Se a tabela tiver *tamanho* posições disponíveis, a função deve ser calculada *h(c) = c % tamanho*, o que ajusta qualquer resultado numérico ao intervalo válido. A escolha do tamanho da tabela como um número primo também produz resultados interessantes. Ou, ainda, é possível

fazer *h(c) = (c % k) % tamanho*, sendo *k* um número primo maior que *tamanho*. Esse tipo de solução é bastante usual para problemas nos quais o conhecimento sobre as chaves é restrito.

Outra abordagem é realizar uma manipulação dos valores das chaves, quebrando-a em partes. Por exemplo, no caso de usar o CPF como chave, é possível pegar cada conjunto de 3 dígitos e somá-los, desprezando os dígitos verificadores. Assim, 123.456.789-77 seria calculado pela soma (123 + 456 + 789) % *tamanho*. Variações permitem inverter algumas das partes, como, por exemplo, calcular o endereço (123 + 654 + 789) % *tamanho* para o mesmo CPF exemplo, com os dígitos centrais pegos em ordem inversa. Outras variações são possíveis, como quebrar o valor de 5 em 5 dígitos, ou mesmo de 2 em 2, ou ainda irregularmente usando 4 e 5 dígitos, por exemplo. Uma vantagem dessas operações é que possuem custo computacional baixo e o cálculo é rápido.

Uma última técnica é fazer o cálculo do quadrado da chave e pegar uma parte dos dígitos resultantes. Por exemplo, usando ainda o CPF 123.456.789-77, pode-se extrair a parte de dígitos significativos, 123.456.789 e calcular seu quadrado, o que resulta em 15.241.57**8.750**.190.521, e deste novo valor, extrair uma parte, obtendo 8750, por exemplo.

Combinações dessas técnicas e de suas variações, bem como outras técnicas, podem ser usadas para avaliar os resultados e se ajustar para conseguir uma boa função de hash.

Tratamento de colisões por endereçamento aberto

Se, ao tentar inserir uma chave em uma posição *e* da tabela, essa posição já estiver ocupada por outro registro inserido anteriormente, existe o caso da colisão. É preciso, então, escolher outra posição livre na tabela para colocar o novo registro e, em uma pesquisa futura, utilizar a mesma estratégia para recuperar a informação. Uma das formas de se tratar colisões é a por *endereçamento aberto*.

O endereçamento aberto corresponde à *sondagem* de uma posição vazia seguindo uma sequência específica.

O primeiro exemplo é a chamada *sondagem linear*. Supondo que, para uma chave *c* seja calculado o endereço h(*c*) = *e*, e que esse endereço já esteja ocupado por um outro registro, usa-se uma nova função rh(*e*) = (*e* + 1) % *tamanho* para determinar o próximo endereço[17]. Assim, a próxima verificação é

17 Na função de recálculo do endereço, rh é uma abreviação para *rehash*, ou seja, "faça o hash novamente".

feita na posição $e + 1$. Se esta também estiver ocupada, aplica-se novamente, obtendo-se $e + 2$. A função é reaplicada até se encontrar uma posição vazia (checar todas as posições da tabela). Lembrando que, se a função hash for uma boa função, a colisão será uma exceção e a sondagem por uma nova posição não deve comprometer o desempenho.

A sondagem linear, na prática, pode usar qualquer incremento, assumindo a forma $rh(e) = e + k$. Porém alguns cuidados têm que ser tomados, pois se *tamanho* for múltiplo de k, nem todas as posições da tabela são sondadas. Por exemplo, se a tabela possuir número par de posições e a sondagem for de 2 em 2, somente as posições pares ou as posições ímpares são sondadas, o que pode prejudicar inserções.

Um problema da sondagem linear é a criação de *agrupamentos*. Um agrupamento corresponde a conjuntos de registros que ficam "juntos" na tabela. Por exemplo, considerando a sondagem linear com k igual a 1, se uma posição i do arranjo estiver vazia, existe uma chance desta ser preenchida caso $h(c) = i$. Porém, se a posição i contiver um registro, a chance da posição $i + 1$ ser preenchida é dada pela chance de $h(c)$ ser igual a $i + 1$, mais a chance de $h(c)$ ser igual i (caso em que a colisão leva ao armazenamento na posição subsequente). Quanto mais posições consecutivas estiverem preenchidas com registros, maior a chance de a primeira posição livre subsequente ser preenchida, pois ela é a única solução para qualquer colisão que ocorra dentro do agrupamento, além da chance de ser indicada diretamente pelo cálculo do hash. O raciocínio é válido para outra k qualquer, diferente de 1, pois os agrupamentos apenas não serão de posições consecutivas, mas de posições separadas de k em k. Esse tipo de agrupamento é conhecido por *agrupamento primário*.

Se a função de sondagem for modificada, os agrupamentos primários podem ser evitados. A estratégia, para essa situação, é fazer com que a sondagem não seja por um valor k constante como na sondagem linear, mas dependente do número de sondagens feitas. Um exemplo dessa alternativa é a *sondagem quadrática*, na qual a sequência de sondagens é dada pela expressão $rh(e) = (e + (-1)^m(m^2/2)) \% tamanho$, na qual a divisão é inteira e m representa quantas vezes a função é chamada ($m = 1, 2, 3...$ *tamanho* – 1). Por essa expressão, os valores gerados são $e + 1$, $e - 1$, $e + 4$, $e – 4$, $e + 9$, $e - 9$... Por uma propriedade prática interessante, se o tamanho da tabela for um número primo que possa ser escrito na forma $4j + 3$, os valores de sondagem gerados incluem todos os valores da tabela. Como um exemplo, se a posição gerada pela função de hash for 6 e a tabela tiver 19 posições (19 é primo e pode ser escrito $4j + 3$, com j igual a 4), as sondagens terão a seguinte sequência: 7, 5, 10, 2, 15, 16, 3, 9, 12, 0, 4, 8, 17, 14, 13, 18, 11 e 1.

Embora se espere que a probabilidade de haver várias colisões em um mesmo endereço seja pequena, é importante notar que, caso isso aconteça, sempre a mesma sequência será utilizada para as sondagens. Quando, para um mesmo endereço inicial de hash, a função de sondagem percorre exatamente a mesma sequência de endereços alternativos, ocorre o problema chamado *agrupamento secundário*.

A eliminação de agrupamentos secundários exige que os caminhos de sondagem sejam diferentes a cada momento. Isto é complicado, pois a pesquisa requer que se saiba o que aconteceu na inserção para que seja possível recuperar a informação. Uma proposta para essa situação é chamada *double hash*, ou *hash duplo*.

No double hash, para uma chave *c* para a qual houve colisão, é utilizada uma função de sondagem rh(*e*) = (*e* + f(*c*)) % *tamanho*. Dessa nova função, f(*c*), deriva o nome "duplo", pois duas funções são utilizadas. O resultado f(*c*) é calculado uma única vez e determina um incremento que depende da chave inserida, e não apenas do endereço onde houve a colisão. Assim, se houve uma colisão para uma chave c_1 em um endereço *e*, sua sequência de sondagens será diferente da de outra chave c_2, desde que f(c_1) seja diferente de f(c_2).

O tratamento de colisões por endereçamento aberto usando double hash não gera nem agrupamentos primários nem secundários. A necessidade de usar essa solução na implementação, é claro, depende do problema que está sendo resolvido e do número de colisões esperadas para o conjunto de dados sendo considerado.

A decisão entre um ou outro modo de resolução para a função de sondagem requer conhecimento sobre a aplicação em si, sobre a quantidade de registros esperados e sobre as características das chaves. Não há uma só solução correta, mas sempre se deve considerar a mais adequada às situações que precisam ser resolvidas.

Tratamento de colisões por encadeamento

Como o endereçamento aberto requer cálculos sucessivos e necessita de alguns cuidados extras em caso de remoções, a solução por encadeamento viabiliza, por meio de um campo adicional, indicar uma lista de colisões.

Supondo que a posição *e* da tabela esteja ocupada e que um novo registro seja direcionado para o mesmo endereço, o tratamento de colisões por encadeamento simplesmente procura por uma posição livre na tabela (que pode ser uma busca sequencial). Uma vez encontrada essa posição e_2, apenas se indica

na posição *e* que seu próximo é e_2. Essa indicação é feita por um campo inteiro, denominado *próximo*, que contém o endereço de quem é o próximo na lista.

A Figura 15 mostra uma tabela com encadeamento para as colisões. As posições vazias são representadas pelo valor -2 armazenado no campo *próximo*. As chaves são indicadas por A_i, B_i e C_i, juntamente com um índice que representa o valor calculado para o hash, de modo que $h(A_i) = i$. Assim, A_5 tem $h(A_5) = 5$ e B_7 representa que $h(B_7) = 7$. Na tabela apresentada, a chave A_1 foi inserida na posição 1 e não possui encadeamento, pois seu próximo é indicado por -1. As chaves A_5 e B_5 possuem hash para a posição 5. Para A_5, a primeira, a própria posição 5 foi utilizada; para B_5, devido à colisão, outra posição foi escolhida (no caso a posição 0). O encadeamento indica que, para h(*c*) = 5, existe a chave na posição 5, seguida pela posição 0, que é indicada no campo próximo. Não existe um próximo para a posição 0, o que é indicado pelo valor -1 no campo *próximo* dessa posição. Para h(*c*) = 7, existem as chaves A_7, seguida por B_7 na posição 10 e por C_7 na posição 3, esta última não seguida por ninguém.

	chave	próximo
0	B_5	-1
1	A_1	-1
2		-2
3	C_7	-1
4		-2
5	A_5	0
6		-2
7	A_7	10
8		-2
9		-2
10	B_7	3
11		-2
12		-2

Figura 15 Exemplo de uma tabela hash que utiliza encadeamento para tratamento de colisões.

O algoritmo que determina uma posição vazia (com *próximo* igual a -2) pode ser qualquer um, pois o encadeamento indica as colisões. Usualmente, esse algoritmo procura sequencialmente, iniciando a pesquisa pela última posição e seguindo em direção ao início do arranjo, o que faz com que as colisões tendam a se concentrar no seu fim.

Um ponto importante sobre a resolução por encadeamento é que as listas devem ser exclusivas para cada hash. Isto quer dizer que, em um encadeamento qualquer, todas as chaves nele necessariamente precisam ter um mesmo h(*c*). Uma implicação importante dessa situação é que, dado o exemplo da Figura 15, se uma chave A_3 precisar ser inserida, a chave existente na posição 3 deve ser movida para outro local, pois a chave C_7 ali presente corresponde a um hash diferente de 3 (basta calcular h(C_7) e ver que vale 7, e não 3). Desta maneira, a chave C_7 deve ser realocada para outra posição vazia para abrir espaço para a chave "legítima" da posição 3, que tem h(*c*) = 3.

Na Figura 16 há uma possível nova configuração para a tabela. A chave C_7 foi movida para a posição 12, tomando-se o cuidado para que a posição 10, que indicava como próximo a posição 3, passe agora a indicar como próximo a posição 12. A posição 3, agora vazia, acomoda a inserção da chave A_3.

	chave	próximo
0	B_5	-1
1	A_1	-1
2		-2
3	A_3	-1
4		-2
5	A_5	0
6		-2
7	A_7	10
8		-2
9		-2
10	B_7	12
11		-2
12	C_7	-1

Figura 16 Modificação da tabela da Figura 15 para acomodar a inserção da chave A_3.

Finalmente, é interessante notar que os mesmos algoritmos usados para as sondagens por endereçamento aberto podem ser usados para achar uma posição livre para o encadeamento. Nesse caso, apenas na inserção ela é usada, pois na pesquisa basta que o encadeamento seja seguido.

Operações usuais sobre tabelas e as tabelas hash

As descrições sobre tabelas hash feitas nos tópicos anteriores concentraram-se em como a tabela funciona e quais suas características de comportamento. As

operações usuais de tabelas acabaram não sendo colocadas explicitamente, o que será feito aqui.

A *criação* de uma tabela hash se inicia com a estruturação de uma tabela vazia, e cada item do conjunto inicial é colocado na tabela usando operações sucessivas de inserção. Desse modo, a criação não é um passo especial, mas uma aplicação de inserções.

A *inserção* é feita sempre usando, inicialmente, a função hash. Havendo colisões, a solução de tratamento escolhida (endereçamento aberto ou encadeamento) deve ser usada, cada uma com suas características.

Operações de *pesquisa* também utilizam a função hash e, a partir daí, as eventuais colisões devem ser sondadas, seja utilizando uma sondagem (endereçamento aberto) ou seguindo-se o campo *próximo* (encadeamento).

Nas *remoções*, no caso de endereçamento aberto, é preciso cuidados para que as sondagens não terminem prematuramente. Por exemplo, simplesmente marcar a posição do item removido como vazia pode fazer com que as sondagens ignorem as colisões posteriores. Nesse caso, marcar a posição como "disponível" (indicando que é diferente de vazia) é uma alternativa. Em relação ao encadeamento, o início da lista deve ser sempre a posição dada pelo hash; assim, a remoção do primeiro registro da lista é feita copiando-se o próximo para a posição do removido e, para os demais, basta ajustar a lista para "pular" o item removido, excluindo-o logicamente do encadeamento.

A tabela hash funciona apenas para pesquisa para a chave escolhida. *Modificações* em campos que não sejam usados na função não têm consequência sobre a estrutura da tabela. Caso a chave seja modificada, usualmente opta-se por uma remoção do registro anterior e uma nova inserção do registro modificado.

Por último, qualquer que seja a solução para o tratamento de colisões, o desempenho da tabela pode ser melhorado por uma operação de *manutenção*. No caso de endereçamento aberto, a eliminação de posições *disponíveis*, tranformando-as em *vazias* diminui o número de sondagens para localizar um registro em uma pesquisa. Para o encadeamento, a realocação dos registros pode diminuir o número de colisões futuras, embora isso não seja muito significativo. Em qualquer situação, a manutenção é usualmente feita colocando-se todos os registros em uma estrutura separada e criando-se uma tabela "do zero", ou seja, definindo-se uma tabela vazia e inserindo todos os registros de volta.

4.4 Considerações finais

A organização de dados em memória principal é muito importante, pois o processador somente tem acesso aos dados nessa memória. Mesmo no caso de dados armazenados em disco, assunto da próxima unidade, considerações sobre como utilizar a memória principal são importantes[18].

Organizações simples, como tabelas não ordenadas, são relevantes pois, possuem baixo custo de inserção e remoção. A pesquisa é seu "calcanhar de Aquiles", com O(*n*) como complexidade de tempo. Um caso interessante sobre tabelas não ordenada ocorre quando alguns registros são mais consultados que outros. Uma técnica conhecida como "mover para o início" pode ser empregada para melhorar a pesquisa. Por essa técnica todo registro pesquisado é trocado com o registro da posição anterior, de modo que, como tempo, os registros mais consultados fiquem mais próximos ao início do arranjo, o que é uma boa opção para a pesquisa sequencial.

O uso de tabelas ordenadas permite um ganho excepcional em relação à operação de pesquisa, usando a pesquisa binária na localização. O custo por essa opção é que tanto as inserções quanto as remoções precisam ser O(*n*).

No caso de tabelas hash há uma grande melhoria em todos os tempos de operações, com inserções, remoções e pesquisas muito rápidas. O problema é a dificuldade de escolha de uma função apropriada para o cálculo do hash, o que pode não ser uma tarefa simples. Adicionalmente, tabelas hash não dão suporte a listagens ordenadas pela chave, o que é de imediato nas tabelas ordenadas e fácilmente nas não ordenadas (as quais podem ser ordenadas em um dado momento, se preciso, sem comprometer as operações).

A decisão por uma ou outra solução, naturalmente, depende das características do problema que deve ser resolvido. Assim, cabe ao desenvolvedor conhecer as características de funcionamento de cada tipo de organização para fazer a melhor escolha.

18 Reveja os comentários sobre a ordenação de dados em disco, na discussão sobre o *mergesort*.

UNIDADE 5

Organização de dados em memória secundária

5.1 Primeiras palavras

Em contraposição à memória principal, dados armazenados em memória secundária sofrem os problemas de latência intrínsecos aos dispositivos externos. No caso de um disco rígido, essas restrições são mecânicas e o tempo de acesso aos dados depende da movimentação do braço de leitura e escrita, além da própria rotação do disco e transferência de dados para a memória principal.

No caso de armazenamento em memória secundária, uma das principais preocupações é a redução do número de acesso a disco. Quando um dado precisa ser recuperado do disco, o acesso é por blocos, o que significa que, na maioria dos casos, vários registros são recuperados de uma vez. Aproveitar essa característica de acesso é importante para que não sejam precisos mais acessos a disco, que os realmente necessários, para executar uma dada operação.

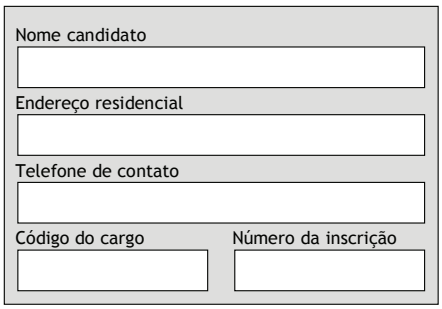
É importante lembrar que o tempo de acesso a disco é muito maior que o tempo de acesso à memória principal. Assim, mesmo nas situações em que se precise trabalhar bastante os dados que estão em memória, o tempo desse processamento tende a ser bem menor que o desprendido para trazer os dados do disco para a memória.

Esta unidade apresenta algumas das principais formas de armazenamento de dados em memória secundária. Em princípio, a mesma visão adotada para a organização de dados em memória principal é seguida. Porém o fato dos dados estarem em disco será considerado e as consequências desse fato serão discutidas.

5.2 Tabelas em memória secundária e restrições de acesso

O ponto de partida para entender as limitações do uso de memória secundária é rever alguns conceitos importantes envolvidos. Inicialmente, é importante que se entendam as relações entre os *registros* e os *arquivos*, para posteriormente vincular a esses conceitos o acesso a disco, caracterizado pelo uso de blocos para acesso de leitura e gravação.

Um *registro* é uma unidade de armazenamento. Em algoritmos, os registros também são chamados de variáveis compostas heterogêneas, dada sua natureza de armazenar um conjunto de informações cujos tipos podem ser diferentes uns dos outros. A Figura 17 ilustra um registro com informações básicas sobre um candidato a um emprego de uma empresa. É possível notar que dois tipos de dados são utilizados (*literal* e *inteiro*), compondo os cinco *campos* necessários.




```
tipo tCanditado:
          registro
            nomeCandidato,
            endereçoResidencial,
            telefoneContato: literal
            códigoCargo,
            númeroInscrição: inteiro
          fim-registro
```

(a) (b)

Figura 17 Exemplo de um registro: (a) Esquema representativo das informações; (b) declaração do tipo de dados.

Registros podem ser utilizados, por exemplo, como variáveis individuais em um algoritmo ou um programa, quando são úteis apenas como organização de dados, o que auxilia na clareza e documentação. O aumento do volume de dados pode ser conseguido também com o uso de vetores de registros (tabelas), ainda no contexto da memória principal.

Quando se deseja guardar um grande número de registros de forma permanente, o uso da memória secundária se faz necessário. Assim, coleções de registros podem ser guardadas em *arquivos*. Um arquivo pode ser visto, do ponto de vista conceitual, como uma sequência de dados; em particular, no enfoque dessa discussão, cada dado individual corresponde a um registro. Um arquivo, então, pode ser considerado como uma coleção de registros, cada qual ocupando uma posição. Por convenção, a numeração das posições se inicia em zero. Um exemplo de um arquivo com quatro registros é apresentado na Figura 18.

O ponto essencial para que se entendam os arquivos como modo de armazenamento é lembrar que estão em memória secundária. Isso significa que não estão diretamente disponíveis para uso em algoritmos ou programas em execução. Para que cada registro fique disponível no contexto do programa, é preciso que seja feita uma operação de leitura, o que implica no acionamento, por parte do sistema operacional, das ações para que os bytes armazenados nos setores de um disco rígido sejam transferidos para a memória principal disponível para o programa, passando pelo *buffer* do sistema.

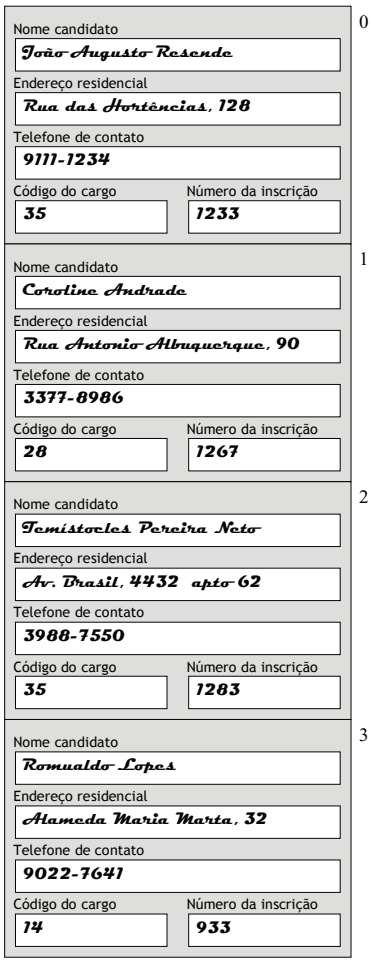


Figura 18 Esquema mostrando um arquivo com quatro registros, ocupando as posições de 0 a 3.

Outros destaques em relação aos arquivos compreendem:

- Diferentemente de tabelas em memória, arquivos não possuem um índice que permite acesso a cada registro. Cada registro deve ser especificado por sua posição e uma operação de leitura deve ser realizada para copiar os dados (bytes) do disco rígido para uma variável na memória principal.
- Os campos não ficam disponíveis separadamente. Usualmente, é necessário que um registro completo seja lido para a memória principal, de forma que somente então um de seus campos possa ser utilizado.[19]
- Alterações de informações exigem leitura do dado para uma variável em memória, modificação dessa variável e posterior armazenamento

19 Recuperação de partes de registros não é uma operação comum. Embora seja possível ser feita, a complexidade do controle necessário para desenvolver um algoritmo nesses moldes aumenta significativamente, sem que se obtenha uma vantagem significativa.

(gravação) do registro de volta no arquivo, sobrescrevendo a informação ali armazenada.

- O tamanho dos arquivos é dinâmico. Quando criado, possui tamanho zero, com nenhum registro. À medida que operações de gravação são feitas no final do arquivo, seu tamanho aumenta. As limitações para o tamanho de um arquivo são dadas pelo sistema operacional e pelo espaço disponível no disco rígido.
- Arquivos não podem ser reduzidos a não ser por *truncamento*, que é uma operação que descarta a parte final do arquivo. Isto significa que não é possível remover do arquivo uma parte intermediária, mas somente a final. Essa operação consiste, em termos práticos, em marcar um dos registros e descartar do arquivo ele e todos seus subsequentes.

Todas essas características aplicam-se aos aspectos *lógicos* de funcionamento de arquivos. Em outras palavras, a visão de um arquivo linear é uma abstração. Quando ligado à parte física do armazenamento de dados em discos rígidos, outros aspectos devem ser considerados. Alguns destes são menos significativos e serão apenas comentados ou até ignorados, outros são importantes e terão o destaque necessário.

O sistema operacional é responsável por mapear um arquivo nos setores (e trilhas e cilindros, por consequência) do disco rígido. Ao fazer isso, toma decisões sobre onde e quando realizar as gravações. Assim, embora um arquivo seja uma estrutura contínua do ponto de vista lógico, o sistema operacional pode mapear vários pedaços dos dados em partes não necessariamente consecutivas do disco[20]. O resultado desse comportamento é o aumento do tempo necessário para leituras posteriores, já que latências adicionais de busca e rotação são envolvidas. Como não há controle do desenvolvedor dos programas sobre esse comportamento, é importante saber que ele existe e quais suas consequências. Não serão feitas, neste texto, considerações de desempenho tão específicas.

O acesso para leitura ou gravação de informações em discos rígidos, dadas as diversas latências envolvidas, possui, como se sabe, tempo de recuperação e armazenamento superiores quando comparados a acessos em memória principal. Essa questão é fundamental quando são considerados registros armazenados em arquivos. O uso de estratégias que considerem a busca de dados em blocos são importantes, visto que pode ser minimizado o número de acessos a disco e, em consequência, o tempo gasto pela execução das operações de manipulação de dados.

20 Neste caso, o termo usado é *fragmentação* do arquivo.

Esta unidade aborda as organizações de arquivos segundo algumas visões importantes, em ordem crescente de complexidade. Para as discussões, são sempre considerados registros de estrutura e tamanho fixos[21], de forma que a localização de um registro dentro dos bytes que compõem um arquivo seja possível.

Em um primeiro momento, são considerados os arquivos sem ordenação e, em seguida, os arquivos ordenados. Essas duas formas de organização, consideradas sob o ponto de vista do conjunto de operações possíveis sobre tabelas, permitem expandir o conceito para uma organização mais sofisticada, que envolve os arquivos indexados.

5.3 Formas de organização

Quando coleções de registros são mantidas em arquivos são obtidas estruturas chamadas de *tabelas em arquivo*, ou simplesmente *tabelas*. Embora o termo seja usado tanto para as estruturas mantidas em memória principal quanto secundária, o contexto permite diferenciá-las quanto a estrutura e manipulação, mas principalmente quanto às formas de acesso. O ponto mais importante de diferenciação, entretanto, é o tempo de acesso à memória secundária ser consideravelmente superior quando comparado à memória principal.

As diversas operações sobre tabelas serão trabalhadas para arquivos sem ordenação, caracterizando os acessos a disco frente às manipulações de dados, firmando os conceitos envolvidos. Em seguida as modificações necessárias para a organização de arquivos ordenados será abordada.

5.3.1 Arquivos sem ordenação

Um arquivo sem ordenação corresponde a um arquivo cujos registros não possuem necessariamente qualquer forma de posicionamento de um registro em relação ao outro em função do seu conteúdo.

Assumindo que se disponha de um conjunto inicial de dados que devem ser usados para formar o arquivo, a operação de *criação* é feita pela criação

21 Os registros mais usuais são registros fixos tanto nos campos quanto no tamanho de cada campo. Porém, é possível que sejam definidos e manipulados registros de tamanhos variáveis, o que envolve registros que podem, ou não, possuir determinados campos (exemplo: se um campo com o número de filhos for nulo, não é preciso ter o campo para o nome dos filhos) ou um mesmo campo ocupar mais ou menos espaço para registros diferentes (exemplo: nomes mais longos ocupam mais bytes que nomes curtos), uu ainda ambos os casos acontecerem simultaneamente. Nessas situações, os controles adicionais sobre o conteúdo do registro e as posições das informações internamente a ele são necessárias. Essas informações são conhecidas por *metadados* e podem existir para o registro, para o arquivo ou para ambos.

de um arquivo vazio, seguida da escrita dos registros, um por vez, no final do arquivo, fazendo com que este último aumente de tamanho a cada novo registro. Considerando-se que o tamanho do problema seja o número de registros inicialmente disponíveis, então a complexidade da criação é O(*n*).

Do ponto de vista conceitual, a operação é simples e consiste em um laço de repetição que faz as diversas escritas no arquivo. A estrutura essencial da lógica de inserção é apresentada no Algoritmo 5-38.

Algoritmo 5-38

```
1  { criação do arquivo não ordenado }
2  associeArquivo(arquivoDados, "nome_arquivo.dat")
3  crieArquivo(arquivoDados)  { cria arquivo vazio }
4
5  leia(númeroInicialRegistros)
6  para i ← 1 até númeroInicialRegistros faça
7     obtenhaRegistro(registro) { digitação dos dados por exemplo }
8     escrevaArquivo(arquivoDados, registro) { escreve novo registro }
9  fim-para
10
11 fecheArquivo(arquivoDados) { encerra o acesso }
```

Há que se considerar, ainda, os aspectos físicos para a criação do arquivo. Cada comando de escrita, antes de provocar o armazenamento efetivo dos dados em disco, faz uma transferência do registro escrito para o *buffer* interno. Somente quando todos os bytes do *buffer* são preenchidos é que a escrita em disco tem efeito. Assim, supondo que em um bloco de disco caibam exatos 50 registros, somente após todos estarem presentes no *buffer* é que o sistema operacional determina um bloco no disco e faz a transferência, em uma única operação, de todo o conteúdo do *buffer*.

É, ainda, possível (e provável) que em um *buffer* não caiba um número exato de registros. Nesses casos, os primeiros bytes do registro ficam contidos em um bloco, enquanto os restantes são alocados no bloco seguinte. Durante as operações de leitura de dados, o sistema operacional se incumbe de "montar" o registro de volta, agrupando novamente os bytes. Nesses casos, a leitura de um simples registro pode exigir dois acessos a disco.

No caso da criação do arquivo, porém, a quebra de um registro em blocos diferentes não é um problema, visto que cada bloco é preenchido e gravado uma única vez, gerando o mínimo de acessos a disco possível.

Os comandos de fechamento de arquivo liberam o sistema operacional de controlar o arquivo, o que faz com que o último bloco, mesmo incompleto, seja escrito efetivamente no disco. Esse último bloco pode ser composto parte por registros válidos e parte por bytes quaisquer, que já estavam no *buffer*. Todos esses bytes são escritos no disco, mesmo o "lixo" ali presente. Como o sistema operacional mantém o controle sobre o número total de bytes válidos, os bytes extras são facilmente ignorados e descartados.

A *inserção* de um novo registro ao arquivo segue lógica similar à da criação, com a escrita do novo item na última posição. Assumindo que o arquivo já esteja aberto para acesso para leitura e gravação, o Algoritmo 5-39 apresenta os passos para uma nova inclusão.

Algoritmo 5-39

```
1   insira(arquivoDados, novoRegistro)
2     { posicionamento no fim do arquivo }
3     posicioneArquivo(arquivoDados, tamanhoArquivo(arquivoDados))
4
5     { gravação do novo item }
6     escrevaArquivo(arquivoDados, novoRegistro)
```

No Algoritmo 5-39, considera-se que a função *tamanhoArquivo* retorne o número de registros; como os registros existentes estão nas posições de 0 a *tamanhoArquivo(arquivoDados)* – 1, o comando *posicioneArquivo* da linha 5.3.1 ajusta o arquivo para escrever após a última posição. Assim, o arquivo é aumentado em um registro, consumindo tempo O(1).

Quando se consideram os acessos efetivos ao disco rígido, é preciso lembrar que todo o bloco que vai conter o novo registro tem que ser reescrito no disco. Para tanto, é preciso, inicialmente, que o bloco seja lido para o *buffer* e manipulado para que passe a conter o novo item e, então, todo o bloco é reescrito de volta no disco. Assim, são necessários dois acessos a disco para que a inserção possa ser feita.

Um ponto importante nesta discussão é que o sistema operacional é o responsável pela leitura e escrita dos dados do *buffer*. Portanto, somente uma leitura será necessária se o bloco ainda não estiver presente no *buffer*. Da mesma forma,

o sistema decide quando o bloco será reescrito no disco, o que pode levar à situação em que várias novas inserções podem ser feitas aproveitando o bloco já lido para o *buffer*, poupando escritas em disco. Não há como prever, do ponto de vista do desenvolvedor do algoritmo, quando as escritas efetivamente ocorrem, mas que pelo menos uma leitura e uma escrita ocorrem é fato.

O procedimento de *remoção* em arquivos não ordenados segue um raciocínio simples para manter os dados do arquivo organizados: o registro da última posição é copiado sobre o registro a ser apagado e, em seguida, o arquivo é truncado de forma a perder a cópia do registro recém-copiado. Essa operação tem complexidade O(1).

Algoritmo 5-40

```
1   remova(arquivoDados, posição)
2     { obtenção do último registro }
3     posicioneArquivo(arquivoDados, tamanhoArquivo(arquivoDados) - 1)
4     leiaArquivo(arquivoDados, últimoRegistro)
5
6     { gravação do registro sobre a posição
7       do registro a ser descartado }
8     posicioneArquivo(arquivoDados, posição)
9     escrevaArquivo(arquivoDados, últimoRegistro)
10
11    { truncamento do arquivo }
12    posicioneArquivo(arquivoDados, tamanhoArquivo(arquivoDados))
13    trunqueArquivo(arquivoDados)
```

Em termos de acesso a disco, essa operação exige a leitura do último bloco do arquivo, onde está o último registro, e também a leitura do bloco em que está o registro a ser removido (caso não sejam o mesmo). Com os blocos disponíveis, a cópia do último registro sobre o removido é feita e o bloco deve ser reescrito em disco. Finalmente, para o truncamento[22], novo acesso a disco deve ser feito para atualizar o último bloco, agora sem o registro final.

22 Pode ser que a operação de truncamento, em muitas linguagens, não possa ser feita com o arquivo aberto. No caso, o arquivo tem que ser fechado, truncado e, então, reaberto. Essas operações consomem tempo e, possivelmente, muitos acessos a disco. Estes não serão considerados no texto.

A *pesquisa* em arquivos não ordenados é feita de forma sequencial, até localizar o registro ou chegar ao final do arquivo, sendo essa última a situação de falha na busca. A busca é feita pela leitura dos registros na ordem de armazenamento, um a um, como ilustrado pelo Algoritmo 5-41.

Algoritmo 5-41

```
1   pesquisa(arquivoDados, chave)
2     { ajusta busca para o início do arquivo }
3     posiçãoAnterior ← posiçãoArquivo(arquivoDados) { preserva
      posição atual }
4     posicioneArquivo(arquivoDados, 0)
5
6     achou ← falso
7     posição ← -1 { contagem atrasada }
8     enquanto não achou e não fimArquivo(arquivoDados) faça
9           leiaArquivo(arquivoDados, registro)
10          achou ← registro.chave = chave
11          posição ← posição + 1
12    fim-enquanto
13
14    { restaura posição anterior do ponteiro do arquivo }
15    posicioneArquivo(arquivoDados, posiçãoAnterior)
16
17    { retorna resultado da pesquisa }
18    se achou então
19          retorne posição
20    senão
21          retorne -1 { indicação de falha }
22    fim-se
```

Em termos de acesso a disco, a busca lê tantos blocos quanto necessários de forma sequencial, examinando os registros contidos em cada um deles. As-

sumindo que todos os registros tenham a mesma probabilidade de serem pesquisados, em média, uma busca bem sucedida varre metade dos registros do arquivo, o que também corresponde à metade dos blocos que formam o arquivo. No caso de busca por um registro não existente, todo o arquivo é analisado e todos seus blocos lidos para o *buffer*. Em qualquer das situações, o tempo é proporcional a O(*n*), sendo *n* o número de registros presentes no arquivo.

As *atualizações* são feitas de forma direta, com a leitura e reescrita do registro no mesmo local, como apresentado no Algoritmo 5-42, o qual assume que a alteração seja feita de alguma forma no procedimento *editaRegistro*. Essa operação é O(1).

Algoritmo 5-42

```
1    altere(arquivoDados, posição)
2     { recuperação do registro }
3     posicioneArquivo(arquivoDados, posição)
4     leiaArquivo(arquivoDados, registro)
5
6     { modificação dos dados }
7     editaRegistro(registro)
8
9     { regravação do registro }
10    posicioneArquivo(arquivoDados, posição)
11    escrevaArquivo(arquivoDados, registro)
```

No caso de atualizações, o bloco com o registro tem que ser lido do disco, os bytes referentes ao registro modificado têm que ser atualizados e, então, o bloco do *buffer* reescrito de volta no disco. Assim, dois acessos a disco são necessários, caso os dados não estejam disponíveis no *buffer* e para garantir que a efetiva escrita no disco seja completada.

Arquivos não ordenados, estruturados da forma descrita, não requerem *manutenção* e, assim, mais nenhuma ação é necessária para definir as operações de manipulação de dados.

As operações descritas referem-se, em termos das considerações sobre necessidades de leitura e escrita dos blocos de disco, a registros que estejam

integralmente contidos em um único bloco. Conforme já comentado, é possível que alguns registros fiquem parte em um bloco, parte em outro. Nesses casos, ambos os blocos precisam ser lidos e atualizados, o que pode gerar, para essas situações específicas, tempo de acesso maior do que para os demais registros.

Finalmente, uma consideração adicional ainda deve ser feita. Da mesma forma que em tabelas em memória, as remoções em arquivos não ordenados podem ser feitas pela simples marcação de um campo de validade do registro. Essa alternativa exige leitura e regravação apenas do bloco que contém o registro removido logicamente. Como outras implicações, a pesquisa deve ignorar os registros inválidos e uma operação de manutenção, que usualmente é feita pela cópia dos registros válidos para um novo arquivo (eliminando assim os marcados como inválidos), e colocando esse novo arquivo no lugar do anterior.

5.3.2 Arquivos ordenados

Arquivos ordenados são, assim como os não ordenados, conjuntos de registros armazenados sequencialmente em memória secundária. Há, porém, a restrição de que a posição relativa dos registros obedeça a um critério de ordem, que é aplicado a uma *chave de ordenação*. Como consequência, as operações sobre as tabelas em disco devem prever que essa ordenação seja mantida e também usufruir das vantagens dela, que é o caso da pesquisa, que pode utilizar o algoritmo de busca binária. Convém ainda lembrar que a ordenação é apenas para uma chave específica e pesquisas por outras chaves que não a de ordenação seguem a operação sobre arquivos não ordenados.

Considerando um conjunto inicial de dados, um arquivo ordenado é *criado* pela colocação de todos os registros em um arquivo sem ordenação e, então, submetendo esse arquivo a uma ordenação apropriada, como é o caso do *mergesort*, procurando mesclar ordenações parciais em memória principal (que são rápidas) com a fusão dessas sequências ordenadas até que se obtenha o arquivo completo ordenado.

Algoritmo 5-43

```
1   { criação do arquivo ordenado }
2   associeArquivo(arquivoDados, "nome_arquivo.dat")
3   crieArquivo(arquivoDados)  { cria arquivo vazio }
4
5   leia(númeroInicialRegistros)
```

```
6   para i ← 1 até númeroInicialRegistros faça
7     obtenhaRegistro(registro) { digitação dos dados por exemplo }
8     escrevaArquivo(arquivoDados, registro)
9   fim-para
10
11  { ordenação }
12  mergesort(arquivoDados)
13
14  fecheArquivo(arquivoDados) { encerra o acesso }
```

Os acessos a disco devem ser minimizados para que a operação consuma menos tempo. Para isso, além do uso da memória principal para ordenações de parte dos dados, o *mergesort* varre cada partição ordenada do início ao fim, de forma que cada registro (e, portanto, cada bloco) seja lido apenas uma vez a cada passo. A complexidade de tempo dessa etapa é vinculada à complexidade do algoritmo de ordenação.

Inserções de novos registros requerem a manutenção da ordenação. Na prática, isso significa que os registros devem ser "movidos" o suficiente para que o novo item, ao ser escrito, ocupe sua posição relativa correta na ordenação. Cada registro, iniciando-se no último, que tiver sua chave de ordenação maior que a do novo registro, deve ser movido uma posição para frente. Uma movimentação de registro significa sua leitura de um local e seu armazenamento na posição subsequente.

O Algoritmo 5-44 apresenta a movimentação dos registros com chave maior que a do novo registro e a consequente escrita do novo registro no local correto.

Algoritmo 5-44

```
1    insira(arquivoDados, novoRegistro)
2     { movimentação dos registros necessários para manter a ordem }
3     posição ← tamanhoArquivo(arquivoDados) - 1 { último }
4
5     terminou ← posição < 0
6     enquanto não terminou faça
```

```
7           posicioneArquivo(arquivoDados, posição)
8           leiaRegistro(arquivoDados, registro)
9
10          se registro.chave > novoRegistro.chave então
11                escrevaArquivo(arquivoDados, registro) { reescreve
                  na próxima posição }
12                posição ← posição - 1
13          senão
14                terminou ← verdadeiro
15          fim-se
16    fim-enquanto
17
18    { escreve novo registro na posição correta }
19    escrevaArquivo(arquivoDados, novoRegistro)
```

Supondo que não haja qualquer tendência quanto aos valores das chaves dos novos registros inseridos, é possível pensar que, em média, metade dos registros é movida para uma inserção. O custo computacional da operação é proporcional a $n/2$ e, assim, $O(n)$.

Quando o último registro é lido, todo o bloco que o contém é transferido para a memória. Quando as leituras e gravações para a movimentação dos registros ocorrem em um mesmo bloco, não há necessidade de que ele seja escrito no disco a todo o momento. É razoável imaginar que as movimentações ocorram todas internamente no bloco e, quando ele estiver pronto, pode ser reescrito modificado no disco rígido. Para que um bloco fique correto após as movimentações, é preciso que um registro (ou parte de um registro) do bloco anterior seja gravado nele. Essa situação leva à leitura do bloco anterior, seguida da movimentação do registro e, então, possibilitando a gravação do bloco. O mesmo procedimento ocorre para o todos os blocos, do fim para o início do arquivo, até que se localize a posição correta para inserção. Em média, metade dos blocos do arquivo precisa ser reescrita.

Uma *remoção* de um registro de uma dada posição, similarmente à inserção, requer a movimentação dos registros. Depois o último registro é eliminado por truncamento, pois ele fica duplicado. Assumindo que todo registro tenha a mesma chance de ser removido, em média metade dos registros tem que ser reescrito e a operação, assim, assume tempo $O(n)$.

Algoritmo 5-45

```
1   remova(arquivoDados, posição)
2     { movimentação dos registros subsequentes }
3     posiçãoÚltimo ← tamanhoArquivo(arquivoDados) - 1
4     para posiçãoAtual ← posição + 1 até posiçãoÚltimo faça
5           { obtém o registro }
6           posicioneArquivo(arquivoDados, posiçãoAtual)
7           leiaArquivo(arquivoDados, registro)
8
9           { reescreve na posição anterior }
10          posicioneArquivo(arquivoDados, posiçãoAtual - 1)
11          escrevaArquivo(registro)
12    fim-para
13
14    { truncamento do arquivo }
15    posicioneArquivo(arquivoDados, tamanhoArquivo(arquivoDados)
16    trunqueArquivo(arquivoDados)
```

Ao se considerar o acesso a disco usando os blocos, é preciso lembrar que um bloco é lido para o *buffer*, as movimentações de registro internas são realizadas e o bloco é regravado em disco. O detalhe de um registro (ou parte dele) ter que ser movido entre blocos também ocorre nessa situação. A regravação do último bloco devido ao truncamento é também necessária, embora possa certamente ocorrer juntamente com as movimentações feitas nesse último bloco.

A *pesquisa* em arquivos ordenados pode empregar a pesquisa binária, cuja ordem de complexidade de tempo é O(log *n*). O algoritmo da pesquisa binária não é apresentado aqui e difere da implementação convencional apenas pela necessidade de, ao se calcular a posição do meio, o registro tem que ser lido antes de se tomar a decisão sobre em qual metade do arquivo a busca deve continuar. A questão dos blocos lidos do disco, por sua vez, merece alguns comentários. Na primeira leitura (registro do meio do arquivo), todo o bloco que o contém é lido para o *buffer* e, caso esse registro esteja dividido em blocos diferentes, ambos devem ser lidos. A cada passo da repetição da pesquisa binária, novo posicionamento e leitura de registro são necessários e, novamente, todo o bloco é lido

para recuperar um único registro. Ao final da pesquisa, quando os "saltos" da busca se tornam menores, é provável que a busca acabe sendo feita dentro de um único bloco, o que evita novos acessos a disco. Apesar dessas considerações, a pesquisa binária ainda é muito eficiente e o custo da manutenção de um arquivo ordenado é recompensado por uma pesquisa muito eficiente.

A *alteração* dos dados de um registro em um arquivo ordenado pode gerar a necessidade de reorganização, o que acontece se a chave de ordenação é o valor modificado. Nesse caso, os deslocamentos são mais uma vez necessários para colocar o registro na posição correta. Em termos práticos, o algoritmo de movimentação deve verificar se a movimentação deve acontecer em direção ao início ou ao fim do arquivo e, um a um, ir movendo os registros até que seja possível colocar o registro modificado em sua nova posição. Considerando que a modificação da chave possa ser uma modificação qualquer, é possível assumir que, uma quantidade de registros proporcional a n deva ser movida em cada alteração, com complexidade resultante O(n). Finalmente, as discussões feitas sobre o uso dos blocos para a inserção e remoção também se aplicam a essa operação.

Já que todas as operações mantêm a tabela ordenada o tempo todo, não se requer a operação de *manutenção*.

5.3.3 Arquivos com controle dos blocos

Das considerações feitas no tópico anterior, é possível notar a possibilidade de que um registro fique dividido entre dois blocos, o que requer, para estes registros, acessos adicionais tanto para leitura quanto para gravação. Por exemplo, se um bloco em disco rígido possuir 4096 bytes (4 KiB) e cada registro tiver 100 bytes, então caberão em um primeiro bloco 409 registros inteiros. O próximo registro terá seus 6 primeiros bytes armazenados no primeiro bloco e os 94 restantes no segundo. O segundo bloco terá ainda 400 registros inteiros e mais 2 bytes iniciais de um registro que será dividido entre o segundo e terceiro blocos e assim sucessivamente.

O gerenciamento da utilização do *buffer* e dos blocos pelo sistema operacional pode, entretanto, ser controlado pelo programador, de forma que uma utilização mais racional do conhecimento dos blocos evite quebras de registro, por exemplo.

Este tópico mostra uma forma de realizar esse controle e os conceitos aqui apresentados serão recuperados nas discussões sobre arquivos indexados.

Como cenário, considera-se um bloco de tamanho b bytes e registros com r bytes cada um, sendo r menor do que b. Calculando-se o maior inteiro resultante

da razão *b*/*r* obtém-se o número de registros inteiros que podem ser armazenados em um bloco. Este valor é representado por *k*. Por fim, ao se calcular $l = b - kr$ consegue-se o número de bytes restantes dentro do bloco. A Figura 19 apresenta uma visão geral sobre o uso do bloco.

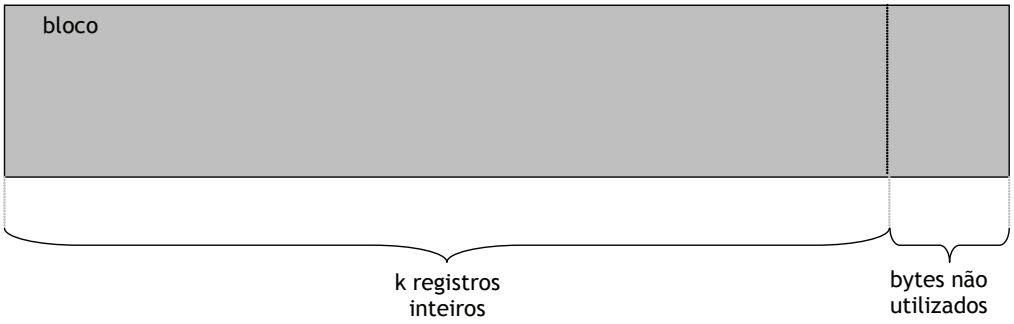


Figura 19 Ilustração de um bloco ocupado parte por registros inteiros e parte não utilizada.

Uma opção para controlar os blocos é definir um arquivo cujos registros tenham exatamente o tamanho de um bloco. Para esse arquivo "especial", segundo o cenário apresentado, cada registro seria composto de um vetor com *k* registros de dados, acrescido de um contador para indicar, dentro do bloco, quantos registros de dados são válidos e um vetor não utilizado, criado apenas para garantir que o número de bytes corresponda ao tamanho de um bloco. O Algoritmo 5-46 mostra uma possível declaração para o registro de bloco.

Algoritmo 5-46

```
1     { registro de controle de bloco }
2     tipo tBloco: registro
3                  dados[k]: tRegistroDados
4                  contador: inteiro
5                  lastro[l]: byte
6             fim-registro
```

A Figura 20 mostra como o registro pode ser usado para armazenar dados. No exemplo, assume-se que cada bloco seja capaz de armazenar 7 registros inteiro (*k*), usando o vetor *dados*. Os *l* bytes restantes no bloco são usados para manter o campo *contador*, além do campo *lastro*, que deve ser dimensionado de forma que o número total de bytes do registro de bloco tenha o tamanho de um

bloco em disco rígido. Na Figura 20 estão armazenados 25 registros de dados, ocupando quatro blocos. O último bloco, ocupado parcialmente, embora armazene efetivamente 7 registros, considera apenas os 4 primeiros, informação dada pelo campo *contador*.

Fazendo-se com que o registro de bloco tenha o tamanho do bloco do disco rígido garante que cada leitura e escrita seja um acesso a disco, nem mais nem menos.

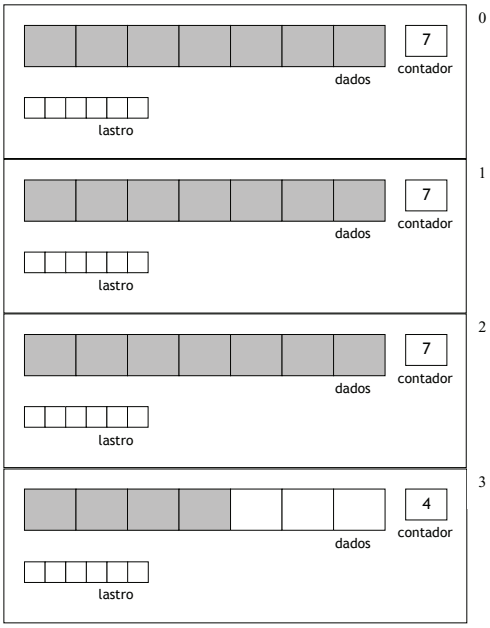


Figura 20 Ilustração de um arquivo com controle dos blocos. O arquivo contém 25 registros de dados distribuídos em 4 blocos. O último bloco é ocupado apenas parcialmente. As posições hachuradas indicam registros de dados válidos.

Se esse controle for usado para armazenar um arquivo não ordenado, por exemplo, a inserção de um novo registro deve ser feita lendo-se o último registro de bloco, colocando o novo registro de dados na quinta posição do vetor e incrementando o contador para 5, e assim, finalmente, regravar o bloco de volta.

Cada comando de escrita ou leitura de um algoritmo teria que ser adaptado a essa estrutura. Esses comandos formariam uma interface para um sistema interno que, conhecendo o tipo da organização do arquivo, decidiria tanto sobre o controle dos dados dentro do registro de bloco como quando haveria necessidade de gravação ou leitura dos blocos do disco.

Usando a Figura 20 como ilustração e supondo um arquivo sem ordenação, uma remoção do registro da posição 8 geraria a seguinte cadeia de eventos:

1. Leitura do registro de bloco 3 para obtenção do último registro (posição interna 3).

2. Determinação da localização do bloco da posição 8, o que é feito sabendo-se que há 7 registros em cada bloco; o resultado é que tal registro é o da posição interna 1 do bloco 1.
3. Leitura do registro de bloco 1 para acesso ao registro desejado.
4. Cópia do conteúdo do registro da posição interna 3 do bloco 3 para a posição interna 1 do bloco 1.
5. Atualização do contador do bloco 3, com decremento do contador.
6. Gravação dos registros de bloco 1 e 3 para atualizar o arquivo em disco.

Outras operações, envolvendo arquivos não ordenados e ordenados devem refletir essa nova estrutura de organização interna e prover os mecanismos adequados para leituras e gravações. Cada operação sobre a tabela em memória secundária, considerando os registros de bloco, passa a se adequar ao novo modelo.

A opção por controlar os blocos é mais complexa, mas permite maior flexibilidade no uso dos arquivos, aumentando as opções de controle restritas do sistema operacional[23]. Uma vez implementada, praticamente quaisquer arquivos que obedeçam a uma mesma forma de organização podem ser utilizados.

5.3.4 Arquivos indexados

Um *índice* é uma estrutura auxiliar, mantida paralelamente ao arquivo de dados, que permite localizar com maior eficiência uma dada informação. Nos termos discutidos nesta unidade, um índice é um arquivo separado, cuja função é auxiliar a localização de um dado registro no arquivo de dados.

Inicialmente é preciso conhecer o que é o índice e quais informações contém. Dada sua função de auxiliar a localização, uma *entrada de índice* corresponde a uma chave (critério usado para a busca) e a sua localização no arquivo de dados. Os índices são mantidos ordenados (física ou logicamente) para permitir pesquisas eficientes.

Por exemplo, supondo um arquivo com dados sobre alunos e assumindo que a chave usada na pesquisa seja o RA, se um registro que contém os dados sobre o aluno de RA 497728 estiver na posição 8135 do arquivo, então o

23 Na realidade, muitos sistemas de gerenciamento de bancos de dados chegam a eliminar completamente o sistema operacional do processo e manter todas as rotinas de acesso a disco internamente. Obtém, assim, um produto diferenciado no mercado, com desempenho e flexibilidade.

arquivo de índice terá um registro contendo o par <497728;8135>. O segundo elemento da entrada de índice é chamando *ponteiro* para o registro de dados. Uma pesquisa é feita no arquivo de índice até que se localize o registro desejado. Com isso, é obtido o ponteiro que dá a localização dos dados efetivos no arquivo de dados. A Figura 21 ilustra essa relação.

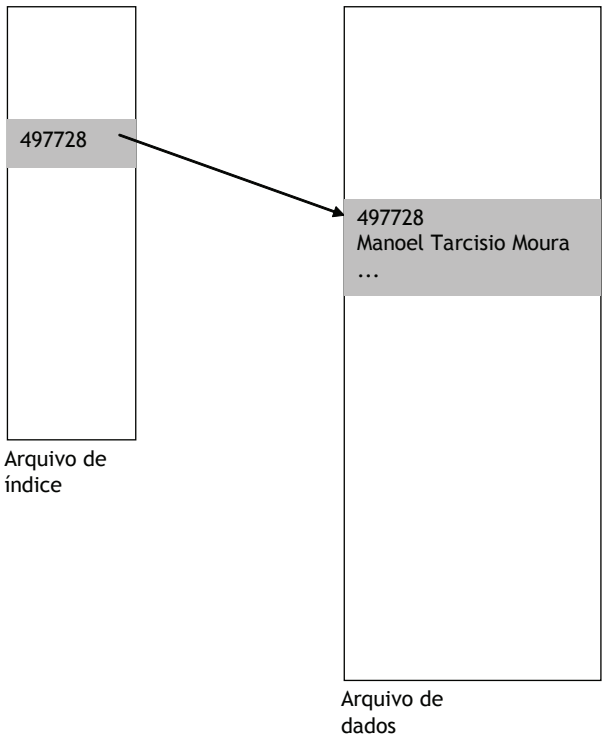


Figura 21 Esquema representando, à esquerda, um arquivo de índice e, à direita, o arquivo de dados. Uma entrada de índice para o RA 497728 está em destaque, juntamente com o registro de dados do aluno. A seta representa a indicação do endereço em que o registro de dados está posicionado. As demais entradas de índice e registros estão omitidas na figura.

Conhecer as vantagens e desvantagens do uso dos índices é importante. Uma desvantagem evidente é a necessidade de mais espaço em disco para manter a estrutura. Outra desvantagem é que, para que as pesquisas no índice sejam eficientes, é preciso que ele esteja ordenado e isso acarreta tempo para manter a organização do próprio índice.

As vantagens, por outro lado, ficarão claras ao longo do texto, à medida que forem sendo descritos os usos dos índices e suas características.

Como primeiro exemplo será apresentado uma suposição sobre um arquivo com um índice associado a ele. Essa estrutura não é uma organização viável (nem muito prática) e tem a função única de mostrar que índices, por si só, já

representam uma vantagem para a pesquisa. Assim, considera-se a seguinte situação:

1. O arquivo de dados é formado por registros com 256 bytes cada um (o que já inclui a chave usada na pesquisa).
2. O arquivo de índice usa uma chave para pesquisa com 10 bytes, além do ponteiro que possui 6 bytes, em um total de 16 bytes por entrada de índice.
3. Todos os registros do arquivo de dados possuem sua respectiva entrada no arquivo de índice, o que leva ambos os arquivos a possuírem exatamente o mesmo número de registros.
4. Ambos os arquivos estão ordenados pela chave usada para pesquisa, o que faz com que a sequência de chaves em um seja idêntica a sequência de chaves no outro.[24]
5. O bloco de disco em questão possui 2048 bytes.
6. Não é feito nenhum controle de blocos explícito, deixando as leituras e escritas para o sistema operacional.
7. Cada arquivo contém 25000 registros.

Dadas essas considerações, é possível mostrar que o tempo necessário para localizar uma dada chave no arquivo de índice é menor que a localização da mesma chave no arquivo de dados. Para tanto, basta fazer alguns cálculos. No arquivo de dados, em cada bloco do disco rígido cabem exatos 8 registros de dados. Como são 20.000 registros no arquivo, são consumidos exatos 2500 blocos em disco. Para o arquivo de índice, cada registro ocupa 16 bytes, de forma que cabem 128 entradas de índice em cada bloco. Para 20.000 entradas de índice são consumido, assim, 157 blocos (156 blocos completos e mais um ocupado 25%).

Supondo que a pesquisa seja sequencial (isto é, um a um a partir do início do arquivo) e que a chave procurada não seja encontrada, o que exigiria que todos os registros tivessem de ser consultados. Nessas condições, a pesquisa no arquivo de índice precisaria de 156 acessos a disco, enquanto a pesquisa no arquivo de dados exigiria 2500 acessos, um valor 16 vezes maior. Mesmo que a pesquisa binária fosse utilizada, o fato de cada acesso a disco disponibilizar um número maior de registros de índice do que de dados, já reduz o número final de leituras realizadas. Além disso, dificilmente uma entrada de índice (que

24 Uma pesquisa binária para achar a chave *C* no arquivo de índice, por exemplo, segue exatamente os mesmos passos quando aplicada para achar a mesma chave *C* no arquivo de dados.

possui somente chave e ponteiro) tem tamanho próximo ao registro de dados (que contém todas as informações relevantes).

Terminada essa apresentação inicial sobre índices e consumo de espaço em disco, na sequencia serão tratadas duas formas de arquivos indexados: os com índice primário e os com índice secundário.

Índices primários

Um índice primário é um índice (que é um arquivo ordenado) para um arquivo de dados também ordenado. Diferentemente do exemplo dado acima, um índice primário não possui uma entrada de índice para cada registro do arquivo principal, mas sim um subconjunto das chaves.

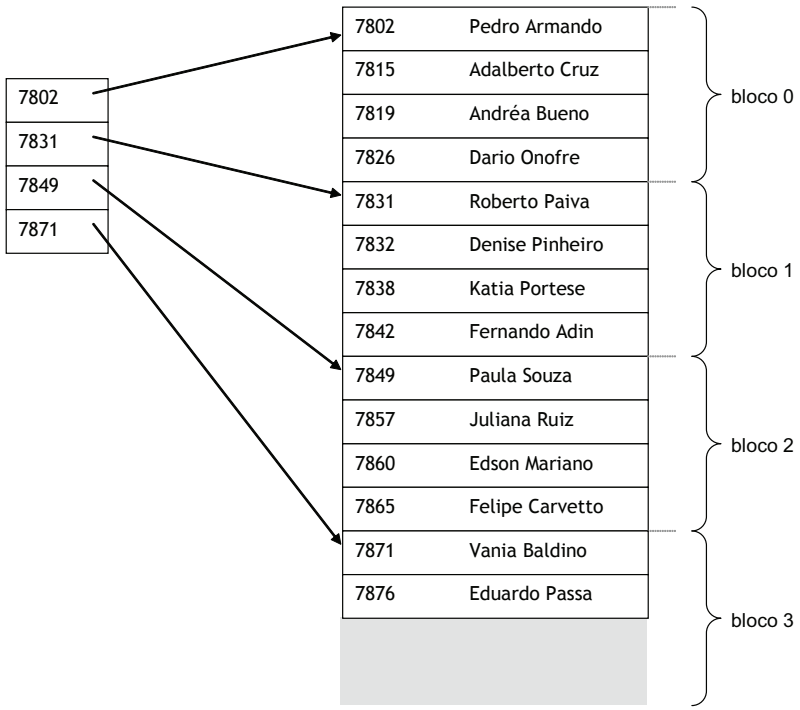


Figura 22 Exemplo de índice primário: arquivo ordenado por código numérico, contendo quatro registros inteiros por bloco, e indexado pelo mesmo campo numérico.

Índices primários são *índices esparsos*, ou seja, possuem número de entradas de índice menor que o número de registros de dados. Isso acontece devido a cada entrada de índice indicar um bloco, e não um registro. Ao conter apenas a primeira chave existente no bloco, o índice tem informações suficientes para localizar qualquer registro.

Por exemplo, a busca pela chave 7849 é iniciada por uma busca no índice, que tem seus blocos lidos para o *buffer*. Por ser um arquivo ordenado, a pesquisa

binária pode ser empregada. A chave em questão é localizada no índice e o ponteiro com o endereço do bloco é utilizado para, em um único acesso a disco, recuperar os quatro registros do bloco. Finalmente o registro com chave 7849 é localizado entre os registros obtidos e a informação desejada é recuperada.

De forma similar, a chave 7860, para ser usada na localização, é procurada no índice. Como não há nenhuma entrada no índice com esta chave, não é difícil verificar que, se o registro com a chave existir, ele necessariamente estará no bloco 2, já que a próxima entrada de índice informa que todas as chaves no bloco subsequente são maiores ou iguais a 7871. O bloco apontado pelo índice é lido para a memória principal e, entre os quatro registros, a informação desejada é recuperada. Caso a chave buscada não existisse no arquivo de dados, então somente um bloco deste arquivo seria recuperado e analisado.

Um exemplo ajuda a compreender as vantagens desta estruturação. Para isso, retomam-se as características dos arquivos no exemplo da seção 5.3.4. Supondo um arquivo de dados com 100.000 registros, seriam ocupados 12.500 blocos de disco. Como deve haver uma entrada de índice por bloco, serão necessários 12.500 registros no arquivo de índice. O arquivo de índice ocupará, então, 98 blocos (sendo o último não completo). Ao se fazer uma pesquisa binária, pode-se grosseiramente dizer que seriam necessárias os $\log_2 98$, ou 6,61, acessos a disco para a busca no índice, embora esse número deva ser menor. Em outras palavras, com um máximo de 7 acessos a disco no índice e mais um acesso no arquivo de dados, qualquer um entre os 100.000 registros pode ser localizado. 100.000 registros com um máximo de 8 acessos a disco torna-se uma boa perspectiva de desempenho.

As vantagens da pesquisa binária e o uso do índice, porém, são penalizadas pelas demais operações. Inserções e remoções no arquivo de dados exigem movimentação média de metade dos registros existentes (no exemplo acima, 6.250 blocos reescritos por inserção ou remoção, em média). É importante lembrar, também, que a cada operação, metade do arquivo de índice também tem que ser atualizado. Esse é um custo alto e tem que ser considerado na escolha por essa forma de organização.

Índices secundários

Uma alternativa para os índices primários são os chamados *índices secundários*. Os índices secundários continuam sendo arquivos ordenados, para permitir uma pesquisa eficiente, mas tiram do arquivo de dados a restrição da ordenação. Assim, índices secundários são índices para arquivos não ordenados.

Uma das vantagens das tabelas em memória secundária serem mantidas não ordenadas é que as inserções (no final) e as remoções (cópia do último registro sobre o removido) são operações de baixo custo se comparadas ao caso dos arquivos ordenados. A desvantagem dos arquivos não ordenados é a pesquisa, que tem que ser sequencial. Ao se usar um índice para um arquivo não ordenado, o problema da pesquisa ineficiente é superado pela rápida pesquisa no índice. Desse modo, o arquivo de dados tem inserções e remoções eficientes, além de contar com uma pesquisa rápida usando o índice.

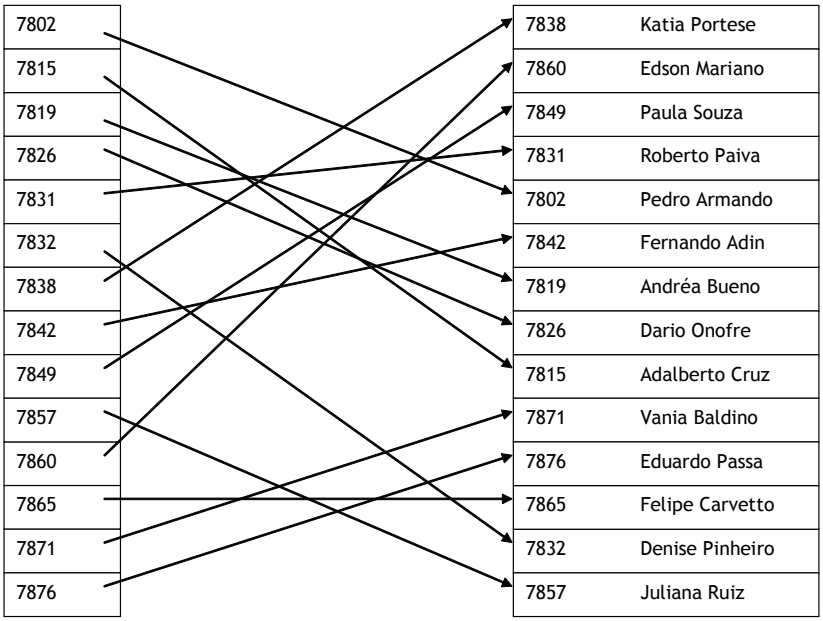


Figura 23 Exemplo de índice secundário: um arquivo não ordenado de dados é indexado por um arquivo ordenado de chaves e ponteiros para os registros individuais.

Porém, o arquivo do índice continua tendo que ser ordenado e sofre a penalização dos deslocamentos de registros quando houver inserções ou remoções de entradas. Outro ponto é que, sendo o arquivo de dados não ordenado, uma entrada de índice por bloco não é mais suficiente para localizar os registros. Desse modo, índices secundários têm que ser *densos*, ou seja, possuir uma entrada para cada registro de dados. A Figura 23 esquematiza um arquivo não ordenado com um índice denso.

Retomando os cálculos para um arquivo de 100.000 registros, nas mesmas condições dos exemplos anteriores, seriam necessários 782 blocos para armazenar as 100.000 entradas de índice. A manutenção do índice ordenado ainda demandaria a movimentação média de 391 blocos em média a cada inserção ou remoção. Por outro lado, o arquivo de dados ocuparia 12.500, mas cada inserção reescreveria apenas o último bloco e uma remoção afetaria apenas o bloco que contém o registro sendo removido e o último bloco do arquivo. No

final de todas as contas, essa organização permite aliar as vantagens dos arquivos ordenados e dos não ordenados, tentando reduzir as desvantagens de ambos.

A principal restrição do uso de índices secundários é manter o índice permanentemente ordenado. Há, entretanto, uma proposição para contornar essa dificuldade. Essa solução gerencia uma estrutura que mantém as entradas de índice organizadas para permitir a pesquisa rápida, mas a ordenação não precisa ser física. Em outras palavras, o arquivo de índice é estruturado com ponteiros, os quais permitem realizar buscas de forma eficiente, mas os blocos apontados podem estar em qualquer parte do arquivo. A estrutura de dados em questão é uma árvore de vários caminhos conhecida como *árvore B*. Neste material não será abordada a estrutura de dados em árvore B.

5.4 Considerações finais

As tabelas armazenadas em disco rígido possuem características semelhantes àquelas mantidas em memória principal. Os aspectos relativos às transferências de dados para o *buffer* interno, porém, devem ser considerados. Saber o que acontece "nos bastidores" é de suma importância para determinar a melhor solução para um determinado problema.

Arquivos não ordenados são práticos e, não possuindo tamanho considerável, podem ser uma boa solução quando há muitas inserções e remoções, sendo a pesquisa não crítica em termos de tempo. Pesquisas em arquivos ordenados, por outro lado, são altamente eficientes, por exemplo, usando a pesquisa binária ou mesmo as pesquisas por estimativa. As operações de inserção e remoção, porém, ficam restritas, pois a ordenação física deve ser mantida.

Os arquivos indexados, sejam estruturados como primários ou secundários, acrescentam maior agilidade às pesquisas, mas o custo de manter a estrutura adicional e o espaço requerido para armazená-la devem ser levados em consideração. Quando a velocidade de acesso é importante, índices são amplamente utilizados.

Os arquivos com índice secundário são uma alternativa importante e, na prática, a forma mais comum de armazenamento em sistemas de bancos de dados. O arquivo de dados é mantido não ordenado, o que o torna prático para inserções e remoções. O índice (que é ordenado por uma dada chave) proporciona o acesso mais eficiente aos registros. Um ponto importante para essa organização é a possibilidade de que índices diferentes podem ser mantidos para um mesmo arquivo não ordenado. Isso significa que pesquisas podem ser feitas para várias chaves diferentes, bastando que para cada novo registro inserido, todos os índices sejam atualizados para suas respectivas chaves.

UNIDADE 6

Conceitos de compressão de dados

6.1 Primeiras palavras

Uma das utilidades mais óbvias para a compressão de dados é poupar espaço em disco. Se uma determinada quantidade de dados pode ser armazenada em um espaço menor, então mais espaço sobrará para outros dados.

As aplicações da compressão de dados, porém, extrapola essa utilidade imediata, pois envolve também a transferência de informações. Para que se obtenha eficiência, quanto mais dados puderem ser transmitidos em um mesmo intervalo de tempo, melhor será a situação. Isso pode ser obtido aumentando-se a velocidade das linhas de comunicação ou, então, reduzindo-se a representação dos dados para que consumam menos da banda de transmissão.

A compressão de dados é uma forma de reduzir a quantidade de símbolos necessários para guardar uma mesma informação, escolhendo símbolos que consumam menos espaço. Em um sistema de controle de uma adega, por exemplo, os tipos de vinho podem ser indicados por “tinto”, “branco” ou “rosé”. Um programador pode usar, por exemplo, uma cadeia de caracteres de 7 bytes para armazenar essa informação, pensando em uma linguagem como C. Essa quantidade de bytes é suficiente para guardar qualquer um dos tipos definidos. Porém a escolha de “T”, “B” e “R” pode representar a mesma informação, usando apenas um byte, o que representa uma economia de espaço pela redução dos símbolos necessários. Pode-se, ainda, ir além, optando-se por usar apenas 2 bits para o armazenamento, com 00 para tintos, 01 para brancos e 10 para rosé (o símbolo 11 não seria utilizado). Em relação à representação inicial, há uma redução de 7 bytes para 0,25 byte para a representação dos dados.

Os símbolos que são escolhidos podem, ainda, não ter comprimento igual para cada informação diferente. Assim, informações mais frequentes podem usar menos bits, enquanto os símbolos mais longos são deixados para os dados que aparecem menos.

Naturalmente, o código deve poder ser revertido. Assim, a partir dos símbolos adotados para representação de um dado, o dado original deve ser obtido de volta[25].

25 Este texto trata da chamada *compressão sem perdas*, que é aquela em que o dado original é obtido do dado comprimido exatamente como estava antes. Alguns métodos de compressão admitem perda de informação, como é o caso da compressão de áudio e vídeo. Quanto maior a compressão de um arquivo MP3, por exemplo, pior a qualidade do áudio e, a partir do áudio comprimido, não é mais possível recuperar a qualidade original.

6.2 Os símbolos e a compressão

A compressão de dados se dá quando, a partir de uma representação de um conjunto, se obtém outra representação equivalente, essa última usando menos espaço. Por exemplo, se existe um arquivo texto, ocupando *s* bytes em uma representação qualquer (ASCII, por exemplo), então uma codificação diferente pode ser utilizada, obtendo-se uma versão comprimida com *t* bytes. O "ganho" obtido com a redução de tamanho pode ser medido pela proporção do número de bytes de diferença entre o arquivo original e o comprimido, proporcionalmente ao tamanho original. Na prática, é calculada a expressão (*s* - *t*)/*s*. Esse valor dá a proporção do ganho e é conhecido como *taxa de compressão*. A compressão, então, ocorre pela substituição de símbolos por códigos (que também são símbolos) de menor comprimento.

Supondo um arquivo texto contendo apenas as letras A, B e C como exemplo inicial, sabe-se que cada letra ocupa 1 byte, se for usada a tabela ASCII como referência. Se o texto contiver um total de 40 letras, então ocupará 40 bytes. Porém, se às letras forem associados os símbolos binários 00, 01 e 10, respectivamente para A, B e C, então, em um único byte poderão ser armazenadas 4 letras (00100101 guardaria ACBB, por exemplo). Usando esse novo código, as mesmas 40 letras ocupariam apenas 10 bytes, com taxa de compressão de 75%. A partir desses 10 bytes "comprimidos" é possível restaurar o arquivo original.

Acrescentando mais uma informação ao exemplo, considere-se que no texto existam 20 letras A, 10 letras B e 10 letras C, em qualquer ordem. Uma segunda opção para escolha de símbolos seria usar o dígito binário 0 para a letra A e os símbolos 10 e 11 para as letras B e C, respectivamente e, assim 01010110 significaria ABBCA. Cada letra A usaria um único bit, de forma que 20 letras A consumiriam o equivalente 2,5 bytes (20 bits). Para as letras B e C, cada uma usaria também 2,5 bytes. A codificação final, portanto, usaria 7,5 bytes[26]. A taxa de compressão, para essa nova codificação, atinge 81,25%.

A Figura 24 mostra ambas as formas de codificação para uma sequência de símbolos. Na Figura 24(a) é apresentada a sequência original de símbolos. Usando-se dois bits para cada símbolo, sendo 00 para A, 01 para B e 10 para C, obtém-se a compressão apresentada na Figura 24(b). Nesse caso, são necessários 5,5 bytes para a representação da cadeia original, de 22 caracteres. Na Figura 24(c) é apresentada a compressão usando-se o símbolo 0 para A, 10 para B e 11 para C, chegando-se a uma cadeia de 4,25 bytes após a compressão.

26 Quando o número de bytes não tem tamanho inteiro, significa que alguns bits do último byte são desprezados por não fazerem parte da representação.

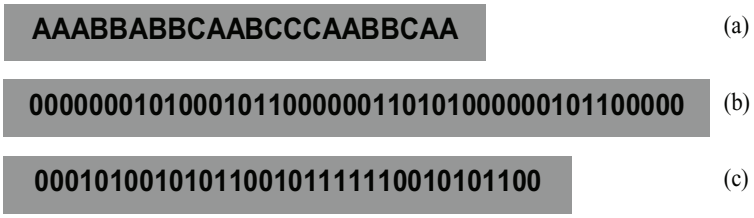


Figura 24 Exemplos de codificação: (a) Sequência original de símbolos; (b) Compressão usando código de 2 bits por símbolo; (c) Compressão usando 1 bit para o símbolo A e 2 bits para os símbolos B e C.

O segredo da compressão de dados é a escolha de novos símbolos que substituam os símbolos originais e que reduzam o tamanho dos dados ao mínimo. Para obter um bom código para tradução de símbolos, alguns pontos devem ser considerados Drozdek (2002):

1. Cada palavra de código deve corresponder a um único símbolo.
2. Durante a interpretação do código comprimido, a interpretação deve ser sequencial.
3. O comprimento de um código de símbolos mais frequentes deve ser menor que o de símbolos menos frequentes.
4. Códigos de menor comprimento não devem ser excluídos do conjunto de símbolos.

O primeiro item é apenas uma resolução de ambiguidade, pois estabelece que cada símbolo original seja mapeado em um único novo símbolo.

Interpretar sequencialmente o código comprimido, por outro lado, é importante para que não se precise “olhar para frente” para tomar uma decisão de como interpretar o código. Em outras palavras, significa que a interpretação do código não pode depender de símbolos que venham depois do símbolo que está sendo interpretado atualmente. Se, por exemplo, o símbolo X tiver como código o símbolo 1, e o símbolo Y for codificado como 10, então, para decidir o significado de uma sequencia iniciada com 1 é necessário olhar o próximo símbolo: se for 0, então significa Y, caso contrário, significa X. Porém, para decidir, é preciso ir para o símbolo seguinte e, depois, retornar para a interpretação. Essa característica é chamada de *propriedade de prefixo*. Isso quer dizer que um dado código nunca é a parte inicial (prefixo) de outro código.

A restrição de que códigos mais longos não sejam associados a símbolos mais frequentes permite codificações de menor comprimento. Se um símbolo A aparece 50 vezes em um conjunto de dados e o símbolo B aparece apenas 5

vezes, então claramente um código menor deve ser associado ao símbolo A e um menor ao B.

Finalmente, não há sentido que códigos curtos não sejam utilizados em uma codificação. Ou seja, se existe um código de menor comprimento sem uso, então ele deve ser empregado no lugar de outro código mais longo. Há, assim, consequente redução do comprimento geral da compressão.

6.3 Codificação de Huffman

O código de Huffman é um procedimento relativamente simples que, a partir da frequência de ocorrência de cada símbolo, gera um código ótimo (HUFFMAN, 1952).

Considerando-se um conjunto inicial de dados, a obtenção do código binário de Huffman é dada pelos seguintes passos:

1. Calcular a frequência de ocorrência de cada símbolo diferente, gerando para ele uma árvore de um único nó, contendo o símbolo e sua probabilidade de ocorrência (dada pelo número de ocorrências dividido pelo número total de símbolos).
2. Criar uma lista com as árvores, em ordem crescente do valor da probabilidade de ocorrência armazenada no nó raiz.
3. Remover da lista as duas primeiras árvores, criando uma nova árvore. Gerar, para isso, uma nova raiz, ficando como ramo direito a primeira árvore da lista e como ramo esquerdo a segunda. Na raiz deve ser inserido o valor de probabilidade dado pela soma das probabilidades de cada uma das duas raízes originais. A nova árvore criada deve ser reinserida na lista, mantendo a ordenação pelas probabilidades de ocorrência. Esse processo deve ser repetido até que a lista contenha uma única árvore (para cuja raiz a probabilidade de ocorrência será 1).
4. Associar o símbolo (código binário) 0 a cada ramo esquerdo da árvore, enquanto o símbolo 1 deve ser associado a cada ramo direito.
5. Criar um símbolo binário, de comprimento variável, para cada símbolo original. Notando-se que cada símbolo original é uma folha da árvore gerada, o novo símbolo binário que deve substituir o original é dado pelos bits da travessia da árvore, iniciando-se na raiz e terminando na folha correspondente.

Como exemplo, considere-se um arquivo texto contendo apenas as letras A, B, C e D. O número de letras A presente no arquivo é 67, o de letras B é 32 e os de C e D, respectivamente, 49 e 93. O tamanho total do arquivo é 241. Assim, as probabilidades de ocorrência dos símbolos A, B, C e D são, respectivamente, 0,278; 0,133; 0,203; e 0,386.

A criação de uma lista com árvores de apenas um nó contendo os símbolos e probabilidades é ilustrado em (a). A lista já se encontra ordenada crescentemente pela probabilidade de ocorrência.

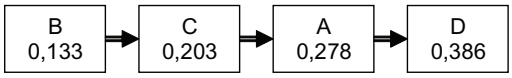
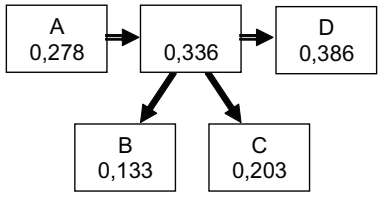
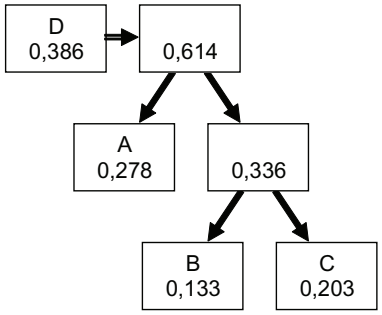


Figura 25 Lista em ordem crescente de probabilidade de ocorrência.

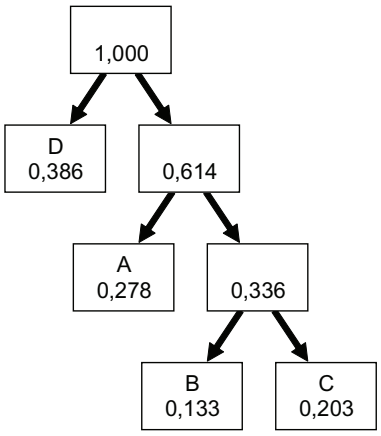
A partir da lista, o procedimento do item 3 descrito acima é aplicado. Inicialmente são unidos os nós de símbolos B e C, criando-se para isso uma raiz com probabilidade 0,336, dada pela soma das probabilidades das duas raízes. Essa nova árvore, reinserida na lista, ocupa a segunda posição, para manter a ordenação. A Figura 26(a) mostra esse primeiro resultado.



(a)



(b)



(c)

Figura 26 Passos da criação da árvore de códigos.

Como há, ainda, três árvores na lista da Figura 26(a), as duas primeiras são unidas em uma nova árvore, com probabilidade acumulada de 0,641. Ao ser inserida na lista, a nova árvore ocupa a última posição. O resultado é apresentado na Figura 26(b), na qual ainda existem duas árvores na lista.

Uma última união reúne os dois itens da lista em uma única árvore, obtendo-se a árvore final, ilustrada na Figura 26(c).

Essa árvore tem associado a cada ramo os dígitos 0 e 1, respectivamente para os ramos esquerdo e direito de cada nó. Essas árvores de códigos é apresentada na Figura 27.

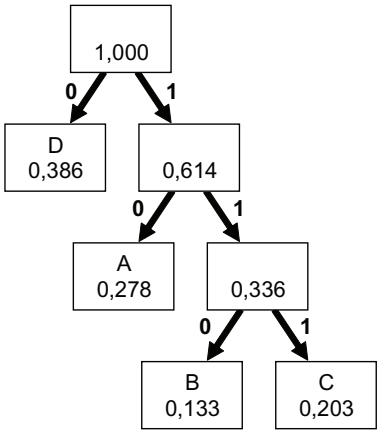


Figura 27 Árvore de códigos obtida pelo algoritmo de Huffman.

O símbolo que deve ser associado, então, a cada letra, é dado pelos dígitos binários dos ramos, seguindo-se o caminho descendente na árvore, da raiz até a folha que contém o símbolo. Dessa forma o símbolo 0 é associado à letra D; o símbolo 10 à letra A, 110 à letra B e, finalmente, o símbolo 111 à letra C.

Para finalizar o processo de compressão, cada letra do arquivo original é substituída pelo código binário derivado pela árvore, gerando uma sequência binária total. Como ilustração do processo, se no texto houver a sequência BCDA, a sequência binária 110111010 a substituirá. A cada 8 bits da saída do processo de codificação será montado um novo byte para o arquivo compactado, sendo que o último byte pode ser "incompleto", ou seja, somente alguns de seus bits podem ser significativos. No arquivo destino devem ser armazenados tanto os códigos novos (comprimidos) quanto a tabela com os símbolos originais, para que a conversão de volta seja possível.

O processo de descompressão dos dados segue o caminho inverso. Dada a tabela de associação dos símbolos originais e dos códigos utilizados, a sequência de bits comprimida é percorrida e interpretada. Assim, reconhecido um zero, o símbolo D é colocado na saída. Caso seja um 1, o próximo valor é consultado; se for 0, então a saída é o símbolo A e, se caso não seja, o próximo bit decidirá entre o B e o C. É importante destacar que os quatro pontos destacados anteriormente como requisitos para um código de compressão são atendidos pela codificação de Huffman.

6.4 Codificação LZW

O algoritmo conhecido por LZW foi proposto por Abraham Lemple, Jakob Ziv e Terry Welch. É caracterizado como um algoritmo adaptativo, ou seja, um algoritmo que se adapta aos dados de entrada para gerar os símbolos de saída. A codificação é uma evolução dos algoritmos LZ77, proposto por Lemple e Ziv em 1977, e LZ78, de 1978.

O princípio de operação do método é substituir sequências de símbolos do conjunto de dados original por códigos de menor comprimento, obtendo assim a compressão dos dados. As sequências de símbolos que são substituídas são armazenadas em uma tabela conhecida como *dicionário*.

O Algoritmo 6-47, adaptado diretamente de Drozdek (2002), mostra um pseudocódigo de alto nível para a compressão pelo LZW.

Algoritmo 6-47

```
1   crie uma tabela com todas as letras
2   crie uma cadeia de entrada s com a primeira letra da entrada
3
4   enquanto houver dados de entrada faça
```

```
5     obtenha o próximo caractere de entrada c
6     se a sequência s + c existe na tabela então
7           s ← s + c
8     senão
9           gere como saída um símbolo para s
10          insira a cadeia s + c na tabela
11          s ← c  { recomeça a cadeia com o novo caractere }
12    fim-se
13  fim-enquanto
14
15  gere como saída um símbolo para s
```

Na prática, para cada símbolo diferente há um código, mas também há códigos para cadeias de símbolos. Quanto maiores forem as cadeias e maior seu número de ocorrências, maior será a compressão, visto que apenas um código mais curto substituirá toda uma série de símbolos.

O dicionário é criado, inicialmente, tendo uma entrada para cada símbolo esperado. Retomando o exemplo do texto que contém apenas as letras A, B, C e D, haverá uma entrada para cada uma destas quatro letras. A Tabela 5 mostra os valores iniciais para esse conjunto de símbolos. A posição é sequencial e indica o código de saída. Como o número de linhas depende dos valores dos dados usados na entrada, o tamanho final do dicionário é, em princípio, indeterminado.

Tabela 5 Tabela que representa o dicionário de símbolos inicial da compressão LZW.

Posição	Símbolo
0	A
1	B
2	C
3	D

A partir da criação inicial do dicionário, a cadeia de entrada pode ser processada. Assim, o algoritmo é seguido e o dicionário recebe mais entradas a partir do momento em que concatenações de símbolos vão sendo acrescentadas.

Como exemplo, pode-se supor que o início do conjunto de dados seja composto pela sequência AABBBAACBBBA. Desse modo, a cadeia de interpretação se inicia com o primeiro símbolo, no caso o A, e é feita a leitura do próximo símbolo, também um A. Como resultado dessa situação, é gerada a saída do código 0 (posição de A na tabela) e é acrescentada ao dicionário a sequência AA na posição 4. Ainda nesse passo, a sequência de entrada é substituída pelo segundo símbolo, coincidentemente um A. O novo ciclo se inicia com a obtenção do próximo caractere, o B. Como AB não está no dicionário, então é feita a saída para o símbolo A (valor 0 novamente), a sequência AB é inserida no dicionário e a cadeia em análise é reiniciada para o símbolo B. O processo se repete para as demais entradas.

A Tabela 6 mostra, passo a passo, o acompanhamento do algoritmo para a criação do dicionário para a cadeia de entrada deste exemplo.

Tabela 6 Dicionário e acompanhamento das entradas e saídas para o processamento da cadeia AABBBAACBBBA para a compressão LZW.

Posição	Símbolo	Entrada	Caractere atual	Saída
0	A	A	A	0
1	B	A	B	0
2	C	B	B	1
3	D	B	B	
4	AA	BB	A	6
5	AB	A	A	
6	BB	AA	C	4
7	BBA	C	B	2
8	AAC	B	B	
9	CB	BB	B	6
10	BBB	B	A	1
11	BA	A	...	

A descodificação segue a interpretação dos símbolos codificados, fazendo a consulta na tabela e gerando de volta os símbolos originais. Naturalmente tanto a tabela quanto a sequência codificada são necessárias para obter a versão completa novamente.

As questões que envolvem esse algoritmo são os tempos de busca na tabela, que pode chegar a tamanhos relativamente grandes. Assim, otimizações que envolvam o tempo de busca e a economia de espaço são envolvidas no algoritmo, mas não fazem parte da discussão deste texto. Uma discussão interessante sobre estes aspectos pode ser vista no livro de Drozdek (2002).

6.5 Considerações finais

A compressão de dados é um assunto importante e, como tal, envolve muito mais que os exemplos citados neste texto. Os algoritmos discutidos envolvem apenas compressão de dados sem perdas, dando uma visão limitada da extensão do assunto. Porém, para compressão de dados sem perdas, tanto Huffman quanto LZW permitem uma visão interessante sobre duas perspectivas diferentes de se obter a redução do volume físico de dados sem perder a informação armazenada.

Muitos programas disponíveis no mercado que fazem compressão de dados utilizam combinações de métodos de compressão diferentes, combinando as vantagens de cada um deles. Essas vantagens e desvantagens devem levar em conta não somente as taxas de compressão obtidas, mas também a complexidade do algoritmo e o tempo que é consumido para comprimir e descomprimir os dados. Em particular, o tempo é importante quando a compressão é utilizada em meios de transmissão de dados, na qual a redução do volume de bytes que devem ser transmitidos não pode ser prejudicada pelo tempo que a compressão leva para ser feita em uma ponta e desfeita na outra.

REFERÊNCIAS

Elmasri, R.; Navathe, S.B. *Sistemas De Banco De Dados*. 4 ed. Pearson Addison-Wesley, 2005.

Folk, M. J.; Zoellick, B.; Riccardi, G. *File Structures: Na Object-Oriented Approach With C++*, 3 ed. Addison-Wesley, 1997. Professor, nestas referências faltam a cidade/lugar onde foram publicados os livros.

Garcia-Molina, H.; Ullman, J. D.; Widom, J. *Implementação de Sistemas de Bancos de Dados*. Campus, 2001.

Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40: 1098-1101, 1952

The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Floating-Point Arithmetic*. New York, 2008.

Sites

Seagate Technology. Ficha técnica SV35.3 Series. Disponível em: <*http://www.seagate.com/docs/pdf/es-ES/datasheet/disc/ds_sv35_3.pdf*>.

SOBRE O AUTOR

Jander Moreira

Obteve sua graduação pela Universidade Federal de São Carlos (UFSCar) em 1989, quando terminou o curso de Bacharelado em Ciência da Computação. Ainda na UFSCar, realizou seu mestrado no Programa de Pós-Graduação em Ciência da Computação, concluído em 1993. O doutorado, finalizado em 1999, foi obtido junto ao Instituto de Física de São Carlos (USP), na subárea de Física Computacional.

Atualmente suas áreas de interesse incluem Processamento Digital de Imagens, Visão Computacional e Redes de Computadores.

Atua como docente junto ao Departamento de Computação da UFSCar desde 1992.

Este livro foi impresso em janeiro de 2011 pelo Departamento de Produção Gráfica – UFSCar.