PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Universidade Federal do Rio de Janeiro

DISSERTAÇÃO DE MESTRADO

**LUCILA MARIA DE SOUZA BENTO**

# Aplicações de Hashing

**Rio de Janeiro**
**2012**

Instituto de Matemática

Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LUCILA MARIA DE SOUZA BENTO

# Aplicações de Hashing

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática e Instituto Tércio Pacciti, Universidade Federal do Rio de Janeiro, como requisito parcial à obtenção do título de Mestre em Informática

Orientadores: Prof. Dr. Jayme Luiz Szwarcfiter
Prof. Dr. Vinícius Gusmão Pereira de Sá

Rio de Janeiro
2012

B475 Bento, Lucila Maria de Souza

Aplicações de Hashing / Lucila Maria de Souza Bento. – 2012.
104 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti de Pesquisa e Aplicações Computacionais (Núcleo de Computação e Eletrônica), 2012.

Orientadores: Jayme Luiz Szwarcfiter; Vinícius Gusmão Pereira de Sá.

1. Hashing – Teses. 2. Complexidade – Teses. 3. Algoritmos Eficientes – Teses. I. Szwarcfiter, Jayme Luiz (Orient.). II. Sá, Vinícius Gusmão Pereira de (Orient.). III. Universidade Federal do Rio de Janeiro, Instituto de Matemática. Instituto Tércio Pacciti. IV Título.

CDD.

Lucila Maria de Souza Bento

# Aplicações de Hashing

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática e Instituto Tércio Pacciti, Universidade Federal do Rio de Janeiro, como requisito parcial à obtenção do título de Mestre em Informática

Aprovado em: Rio de Janeiro – 01 de agosto de 2012

_____

Prof. Dr. Jayme Luiz Szwarcfiter – UFRJ(Presidente)

_____

Prof. Dr. Vinícius Gusmão Pereira de Sá – UFRJ

_____

Prof. Dr. Edson Norberto Cáceres – UFMS

_____

Profa. Dra. Lilian Markenzon – UFRJ

_____

Prof. Dr. Mitre Costa Dourado – UFRJ

Rio de Janeiro
2012

*Aos meus pais.*

# AGRADECIMENTOS

Primeiramente, agradeço aos meus pais pelo exemplo de perseverança, educação e apoio que me deram durante toda a vida.

Ao meu marido Higor pelo carinho, amor, incentivo e compreensão pelos vários momentos em que não estive presente para lhe dar a atenção merecida.

Aos professores Jayme e Vinícius pelo excelente trabalho de orientação, apoio, atenção e incentivo quem foram de extrema importância para a conclusão deste trabalho. Em especial, pela confiança em mim depositada.

Aos colegas e amigos dos laboratórios AC e Projetos, com os quais tive o prazer de conviver nestes últimos anos.

Aos professores e funcionários do Programa de Pós-Graduação em Informática da Universidade Federal do Rio de Janeiro que de várias formas contribuíram para a conclusão deste trabalho.

Ao CNPq pelo apoio financeiro.

E principalmente a Deus, por ter colocado todas essas pessoas em meu caminho.

# RESUMO

BENTO, Lucila Maria de Souza. **Aplicações de Hashing**. 2012, 104f. Dissertação (Mestrado em Informática) – Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2012.

A organização de dados em tabelas de dispersão é uma técnica bastante difundida, onde os valores a serem armazenados estão relacionados a chaves de busca usadas para o cálculo dos endereços de seu armazenamento. Por proverem bom desempenho nas chamadas operações de dicionário (inserção, busca e remoção) sem requerer espaço exorbitante, as tabelas de dispersão são empregadas frequentemente na indexação de grandes volumes de dados e em aplicações típicas relacionadas à associação, busca e manipulação da informação. No entanto, há problemas que, mesmo possuindo natureza bem diversa, podem ser solucionados de maneira elegante com o emprego de tabelas de dispersão, resultando em sensível economia de tempo e espaço. Este trabalho ilustra o poder dessa técnica, também chamada *hashing*, em situações que não lhe são comumente associadas.

**Palavras-chave:** hashing, complexidade, algoritmos eficientes.

# ABSTRACT

BENTO, Lucila Maria de Souza. **Aplicações de Hashing**. 2012, 104f. Dissertação (Mestrado em Informática) – Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2012.

The organization of data into hash tables is a well-known technique in which the persistent values correspond to search keys upon which their storage addresses are calculated. Since they provide good performance for the so-called dictionary operations (insertion, search, deletion) without requiring exorbitant space, hash tables are very often employed in the indexation of large amounts of data and in typical applications related to search and handling of information. Nevertheless, there are problems of rather different nature that can be solved in elegant fashion by employing hash tables, yielding significant economy of time and space. This work illustrates the power of such technique in situations it is not usually associated to.

**Keywords:** hashing, complexity, efficient algorithms.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| CCS | Compressed Column Storage |
| CM | Custo Médio |
| CRS | Compressed Row Storage |
| LRU | Least Recently Used |

# SUMÁRIO

# 1 INTRODUÇÃO

A origem do conceito de *hashing* é incerta. Estima-se que o primeiro texto a apresentar a ideia date de meados de 1950 (KNUTH, 1998). Existem hoje bons textos cobrindo os aspectos teóricos desta que é uma técnica empregada em um número expressivo de softwares de ponta. No entanto, percebemos que existe uma indesejável lacuna entre os dois mundos. Por um lado, a indústria a utiliza porque reconhece os evidentes ganhos em desempenho que são obtidos em muitas situações, ainda que ignore seu funcionamento exato e confie, portanto, em soluções fechadas, importadas de bibliotecas-padrão, e cujo desempenho é aferido empiricamente. Já os acadêmicos conhecem a ciência por trás da técnica, mas talvez pequem ao não expor de forma suficientemente didática, abrangente, ou mesmo enfática, seu enorme poder e aplicabilidade.

Além disso, a técnica de *hashing* é muito utilizada para resolver problemas que envolvem dicionários, mapeamentos explícitos do tipo chave-valor. Essa talvez seja sua principal aplicação, sendo certamente a mais conhecida. Neste trabalho são apresentados problemas de naturezas diversas, de forma a mostrar que a técnica de *hashing* pode compor soluções simples e eficientes de problemas que a princípio não sugeririam seu uso.

## 1.1 Organização do Trabalho

Este trabalho está organizado como segue.

O Capítulo 1 apresenta o assunto e objetivos da dissertação, bem como sua organização e os conceitos básicos utilizados.

No Capítulo 2 estão reunidos os aspectos teóricos referentes à técnica de *hashing*, como funções de dispersão e métodos de tratamento de colisões. Além de, a nível de completude, fazer uma breve apresentação do *hashing* universal, a ideia pouco difundida do *hashing* perfeito e *hashing* dinâmico.

O Capítulo 3 oferece soluções para problemas de naturezas bastante distintas. Alguns são bem conhecidos e contam com um bom número de soluções publicadas; outros são menos conhecidos, ou mesmo novos, assim como as soluções que apresentamos. Para cada problema é apresentada ao menos uma solução com *hashing* e uma que não utiliza a técnica, acompanhadas de uma análise comparativa.

No Capítulo 4 são apresentados os resultados referentes aos tempos de execução da solução de alguns dos problemas estudados, ilustrando assim o poder da técnica de *hashing* por trás de cada solução proposta.

Finalmente, o Capítulo 5 apresenta uma conclusão do que foi apresentado neste trabalho e tece algumas considerações finais.

## 1.2 Definições básicas e notações

Um algoritmo é uma sequência de passos ordenados e bem definidos que descrevem como resolver um problema. Normalmente, para facilitar a descrição e compreensão dos passos, é realizada a modelagem dos dados através de uma estrutura de dados, que é nada mais do que uma forma (eficiente) de armazenamento e

organização de dados em um computador.

*Funções de complexidade* $f(n)$ são usadas para medir os custos computacionais relacionados à execução de um algoritmo para uma entrada de tamanho $n$. Os custos podem ser espaciais, quando relacionados à quantidade de memória necessária à execução do algoritmo; e temporais, quando representam o número de instruções necessárias a sua execução.

As análises de complexidade estão relacionadas às possíveis instâncias de entrada de um problema. As análises de pior caso estão relacionadas às entradas que demandam maior custo computacional, isto é, as mais desfavoráveis possíveis, ao passo que as análises de caso médio retratam o comportamento esperado do algoritmo, quando são consideradas todas as entradas possíveis e suas probabilidades de ocorrência (esperança matemática).

Em Computação, a notação $O$ é comumente empregada para expressar limites superiores assintóticos para as funções de complexidade, a notação $\Omega$ para expressar limites inferiores assintóticos e a notaçao $\Theta$ é usada para expressar limites assintóticos justos (apertados).

## 1.3 Complexidade de algoritmos

Considere um problema $A$. Dizemos que $A$ é um problema de decisão se cada uma de suas instâncias admite apenas *sim* ou *não* como resposta. Além disso, $A$ é polinomial se existe (pelo menos um) algoritmo polinomial para resolver qualquer uma de suas instâncias. A classe $P$ é formada pelos problemas de decisão polinomialmente limitados.

Se $A$ admite um algoritmo que verique uma solução para o *sim* em tempo polinomial no tamanho da entrada, dizemos que $A$ pertence à classe $NP$. Assim,

temos que $P \subseteq NP$ , mas não se sabe se $NP \subseteq P$.

Um problema $A$ é *NP-difícil* se todo problema de decisão $B \in NP$ puder ser polinomialmente reduzido a $A$. Ou seja, se existe um algoritmo que transforme qualquer instância $y$ de $B$ em uma instância $x$ de $A$, de forma que a resposta para $y$ fornece resposta para $x$. Além disso, se $A \in NP$, dizemos que $A$ é *NP-completo*.

# 2 HASHING

A técnica conhecida como *hashing* pode ser considerada um método de pesquisa e organização física de tabelas. Consiste, basicamente, no uso de uma função para realizar o mapeamento de chaves em posições de tabela, mantendo as informações espalhadas de acordo com uma lógica que facilitará as buscas, diferente de estruturas como pilhas, listas e filas, por exemplo, em que as informações são mantidas de forma estritamente sequencial. Outros termos encontrados na literatura incluem *cálculo de endereço* (WIRTH, 1985), *espalhamento* (TENENBAUM; LANGSAM; AUGENSTEIN, 1990) e *tabela de dispersão* (SZWARCFITER; MARKENZON, 1994). A este propósito preferiremos, ao longo do texto, o uso deste último termo em detrimento do termo em inglês *hashing*, a fim de consolidar termos computacionais em português.

Para ilustrar a relevância do método, considere um conjunto de chaves inteiras de 1 a $n$. Se fizermos uso de uma tabela de tamanho $n$, será possível armazenar cada chave $i$ do conjunto na posição $i$ da tabela, e qualquer chave poderá ser acessada, evidentemente, em tempo constante. Este procedimento é conhecido como *acesso direto* (SZWARCFITER; MARKENZON, 1994). Por outro lado, suponha que este conjunto seja muito pequeno (contenha, digamos, apenas 10 elementos), mas cada cada chave possa ser um número com 4 algarismos (de 0 a 9). Para essa

entrada seria necessário alocar uma tabela composta por 10000 posições, $99,9\%$ das quais não seriam sequer utilizadas! Por esta razão, uma função um-para-um para a determinação do endereço da chave, como a do acesso direto, não é geralmente indicada.

Utilizando tabelas de dispersão, é possível utilizar uma quantidade de memória *proporcional à quantidade de chaves que serão armazenadas*, e não à cardinalidade do conjunto de todas as chaves possíveis, conseguindo, ainda assim, um tempo de acesso constante na média dos casos.

Na construção de uma tabela de dispersão, aloca-se memória para uma tabela da forma usual (utilizando vetores, ou *arrays*). Seu funcionamento, no entanto, é tal que a posição que cada elemento irá ocupar, quando de sua inserção na tabela, é *calculado* a partir de sua chave, sendo a imagem dada para aquela chave por uma assim chamada *função de dispersão*. Sendo o processo de busca similar ao de inserção, o cálculo efetuado sobre a chave desejada resultará na mesma posição calculada anteriormente, e o elemento a ela associado poderá ser recuperado.

A figura 2.1 mostra o mapeamento de um conjunto de informações composto pelo par $(chave, valor)$ em uma tabela de dispersão. A função de dispersão utilizada é $h(x) = x \bmod m$, onde $x$ é a chave a ser mapeada, $m$ é o tamanho da tabela de dispersão e *mod* retorna o resto da divisão inteira de $x$ por $m$.

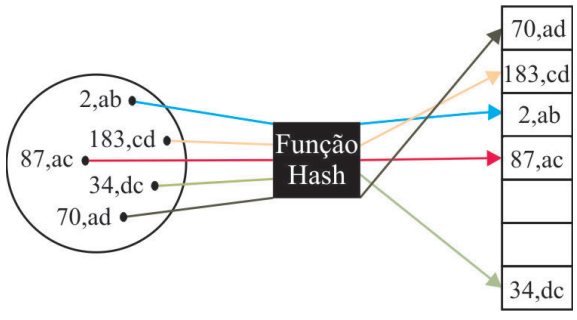


Figura 2.1: Armazenamento com *hashing*

## 2.1 Função de dispersão

Uma função *hash* ou função de dispersão realiza o mapeamento das chaves em posições da tabela de dispersão. Para uma tabela de dispersão de tamanho $m$, uma função de dispersão gera para cada elemento $x$ de um conjunto de $n$ chaves um endereço-base $h(x)$, entre 0 e $m-1$, correspondente a uma posição da tabela (KNUTH, 1998).

Para uma boa função de dispersão $h$, a probabilidade de que uma chave $x$ seja mapeada para cada endereço-base $k$ da tabela deve ser uniforme. Isso significa que, para uma tabela de dispersão de tamanho $m$, e para toda chave $x$ do universo de chaves considerado e para todo endereço-base $0 \leq k \leq m-1$ da tabela, a probabilidade de $h(x)$ ser igual a $k$ deve ser tão próxima a $\frac{1}{m}$ quanto possível (CORMEN et al., 2001). No entanto, mesmo que se obtenha uma função de dispersão que distribua os elementos de maneira absolutamente uniforme ao longo do espaço da tabela, existe uma probabilidade positiva de que chaves com valores distintos sejam mapeadas para o mesmo endereço-base, gerando o que é chamado de *colisão*. Com efeito, Feller descreveu em 1968 o famoso "paradoxo do aniversário" (GEHAN, 1968) , onde, num grupo de 23 ou mais pessoas juntas ao acaso, existe uma chance ligeiramente maior do que 50% de que duas daquelas pessoas comemorem aniversário no mesmo dia do ano. Sendo assim, se for utilizada uma função de dispersão perfeitamente uniforme que mapeie não mais que 23 chaves em uma tabela de tamanho 365, já é mais provável haver do que não haver alguma colisão ali.

A função de dispersão deve ser determinística (SZWARCFITER; MARKENZON, 1994), ou seja, sempre retornar o mesmo endereço-base quando aplicada a uma determinada chave. Para que seja possível cumprir este requisito, funções que dependam de parâmetros variáveis como a saída de um gerador de números pseudo-aleatórios, a hora do dia, o endereço atual de memória do elemento que está sendo vislumbrado etc. são excluídas, pois esses valores podem mudar durante a execução

do algoritmo em questão, alterando o endereço obtido pela função de dispersão para uma chave mapeada anteriormente e tornando impossível sua eficiente localização.

O custo da computação de uma função de dispersão deve ser pequeno, constituído por poucas operações básicas (WIRTH, 1985), para que uma solução baseada em tabelas de dispersão seja mais eficiente do que outras abordagens. Por exemplo, uma árvore rubro-negra pode localizar um item em uma tabela de $n$ itens em tempo $O(\log n)$ (SZWARCFITER; MARKENZON, 1994). Uma tabela de dispersão será mais eficiente do que uma árvore rubro-negra, se sua função de dispersão puder ser calculada em menos tempo (e produzir um número não tão grande de colisões).

A seguir revisitamos as funções de dispersão mais conhecidas.

### 2.1.1 Método da Divisão

O método da divisão é uma das funções de dispersão mais simples e mais utilizadas, na qual uma chave inteira $x$ é mapeada para uma das $m$ posições da tabela de dispersão tomando o resto da divisão de $x$ por $m$ (CORMEN et al., 2001):

$$h(x) = x \bmod m.$$

É possível obter um bom espalhamento das chaves com este método. Há situações em que é necessário o cuidado adicional de se selecionar um valor adequado para o tamanho da tabela. Por exemplo, algumas vezes pode ser conveniente escolher um $m$ par, mas isso implica que se $x$ é par $h(x)$ é par, e se $x$ é ímpar $h(x)$ é impar. Se as chaves são equiprováveis, esse fato não representa um problema, mas se as chaves pares são mais prováveis que as chaves impares (ou o contrário), fica claro que a função não irá distribuir os elementos uniformemente. Outra escolha geralmente infeliz para $m$ seria uma potência de dois, isto é, $m = 2^k$ para algum inteiro $k > 1$.

Apesar da simplicidade, essa é uma função indesejável já que a divisão de um número na sua representação binária por $2^k$ é simplesmente o deslocamento de seus *bits* em $k$ casas para a direita. Com isso, o resto da divisão deste número por $2^k$ é na verdade a extração dos $k$ *bits* menos significativos do número. Para evitar essa situação, o ideal é a escolha de $m$ primo que não seja próximo de nenhuma potência de 2, ou um $m$ que não possua divisores primos menores que 20 (SZWARCFITER; MARKENZON, 1994).

### 2.1.2 Meio do Quadrado

Seja $x$ uma chave inteira. Neste método, a chave $x$ é multiplicada por ela mesma e o endereço-base é obtido pela seleção de um número $r$ de dígitos do meio do resultado, gerando valores no intervalo de 0 a $2^r - 1$. Este método funciona bem, pois a maioria dos *bits* da chave (ou mesmo todos) contribuem para o resultado (DROZDEK, 2001). Por exemplo, considere os elementos cujas chaves são números de 4 dígitos. O objetivo é determinar endereços-base para uma tabela de tamanho 100 (ou seja, num intervalo de 0 a 99). Esse intervalo é equivalente a dois dígitos da chave. Isto é, $r = 2$. Se a entrada é o número 5132, então o quadrado desta entrada é um número de oito dígitos, 26337424. Como precisamos de dois dígitos para compor o endereço-base de cada chave, então o quarto e o quinto dígitos, digamos, podem ser selecionados, resultando no endereço-base 37 para a chave 5132.

É importante observar que após definir as posições, na chave, dos dígitos que serão extraídos para compor os endereços-base, essas posições serão sempre utilizadas, para todas as chaves.

### 2.1.3 Método da Dobra

Neste método, as chaves são interpretadas como uma sequência de dígitos escritos numa tira de papel. Ele consiste em "dobrar" esse papel de $j$ em $j$ dígitos (onde $j$ é o tamanho do endereço-base) de maneira que os dígitos se sobreponham. Depois de dobrar, estes dígitos devem ser somados sem considerar o "vai um". Vale destacar que o método da dobra também pode ser usado com números binários. Nesse caso, ao invés da soma, deve-se realizar uma operação de "ou exclusivo", pois operações de "e" ou de "ou" tendem a produzir endereços-base concentrados no final ou início da tabela (SZWARCFITER; MARKENZON, 1994). Por exemplo, suponha que queremos armazenar a chave 3456 em uma tabela de dispersão com tamanho igual a 100. Utilizando o método da dobra, o endereço-base desta chave é facilmente obtido efetuando $3 + 6 = 9$ e $4 + 5 = 9$, que gera o endereço 99 para a chave 3456.

Inúmeras combinações de dígitos podem ser realizadas com esse método, sendo que nem todos os dígitos devem necessariamente ser utilizados na operação. Por exemplo, os dígitos das chaves que se repetem com muita frequência no universo de entrada podem ser deixados de fora para otimizar o processo.

### 2.1.4 Transformação de base

No método de transformação de base uma chave $x$ tem sua base numérica modificada, resultando em uma sequência de dígitos diferente (DROZDEK, 2001). Por exemplo, suponha uma chave $x$ igual ao decimal 245, que transformado em base octal resulta 365. Este valor é então dividido por $m$ (tamanho da tabela de dispersão) e o resto da divisão é usado como endereço-base de $x$ na tabela.

Como em qualquer outro método, no método da transformação de base as colisões também não podem ser evitadas, uma vez que dividendos diferentes podem

produzir o mesmo resto. Por exemplo, se $m = 100$, embora 245 e 501 sejam diferentes nas bases decimal e octal (365 e 765, respectivamente), ao dividir ambos por 100 obtêm-se o mesmo endereço-base, que é 65.

### 2.1.5 Análise dos Dígitos

No método da análise dos dígitos, o endereço-base é obtido através da seleção de alguns dígitos da chave. O número de dígitos a serem selecionados depende do comprimento do endereço-base desejado. Este método está baseado na análise do universo de chaves para que possam ser determinadas quais são as posições, nas chaves, dos dígitos que devem ser extraídos para a formação dos endereços-base. Isto é feito analisando-se cada um dos possíveis conjuntos de posições com a cardinalidade desejada, de forma a se determinar aquele que produz as distribuições de dígitos mais uniformes.

Como exemplo, suponha que um conjunto de 5000 chaves de 10 dígitos é analisado para determinar a frequência com que cada dígito aparece em cada posição da chave. Suponha também que para formar o endereço-base são necessários 4 dígitos, e que as posições 2, 3, 7 e 9 produzem a distribuição mais uniforme de dígitos, sendo então selecionadas. Neste caso, se a chave 3219046578 for considerada, ela será transformada no endereço-base 2167. É também comum inverter-se a ordem dos dígitos selecionados, donde aquela mesma chave resultaria no endereço-base 7612.

O uso deste método é recomendado a conjuntos de chaves que não mudam ao longo do tempo, sendo factível quando o conjunto de chaves é pequeno, já que é necessário conhecer todas as chaves de antemão.

Neste e nos métodos apresentados anteriormente, assume-se que os valores das chaves são inteiros, mas esta não é uma restrição, uma vez que, naturalmente, qualquer valor de chave pode ser representado por uma sequência de *bits*. O que

é exigido pela maioria das implementações de tabelas de dispersão é que as chaves sejam objetos imutáveis na linguagem de programação utilizada.

## 2.2 Tabela de dispersão

A tabela *hash*, ou tabela de dispersão, é, portanto, a estrutura de dados que armazena os elementos. Tem como objetivo possibilitar que, a partir de uma chave, seja realizada uma busca rápida para se localizar e obter um valor que lá esteja armazenado.

O tamanho $m$ de uma tabela de dispersão é determinado com base no tamanho do conjunto de chaves que serão ali armazenadas (e não, como já mencionado, no tamanho do universo de todas as possíveis chaves), utilizando uma função de dispersão para transformar cada chave em um valor no intervalo de $[0, m-1]$.

Há aplicações em que o tamanho da tabela é modificado dinamicamente, exigindo tratamento especial. Algumas técnicas que levam em consideração um tamanho variável de tabela foram desenvolvidas, dentre as quais pode ser destacado o *hashing dinâmico*, que será apresentado mais adiante.

## 2.3 Tratamento de Colisões

Como a tabela de dispersão é menor que o universo da entrada, uma função de dispersão não pode ser injetora, podendo gerar, ao longo da execução do programa, um mesmo endereço-base para chaves distintas, caso este que é chamado *colisão*. É necessário, portanto, o uso de métodos para organizar e acessar os elementos colididos, chamados métodos de tratamento de colisão (SZWARCFITER; MARKENZON, 1994), cujo desempenho depende fortemente do fator de carga da

tabela, definido como

$$\alpha = \frac{n}{m},$$

onde $n$ é o tamanho do conjunto de chaves armazenadas e $m$ é o tamanho total da tabela de dispersão. Isto se dá porque, mesmo em casos em que são utilizadas boas funções de dispersão, à medida em que o fator de carga aumenta, a probabilidade de haver colisões também aumenta. Como regra empírica, não é indicado fator de carga superior a 75%.

### 2.3.1 Tratamento de Colisões por Encadeamento

No tratamento de colisões por encadeamento, as chaves colididas são armazenadas em listas encadeadas externas à tabela, utilizando memória adicional àquela alocada para a tabela, ou internas, utilizando posições da própria tabela de dispersão.

#### *2.3.1.1 Encadeamento Exterior*

Nesse método de tratamento de colisões, cada posição da tabela de dispersão armazena, de fato, o nó-cabeça de uma lista encadeada. Cada nó da lista encadeada, além do campo para armazenar o ponteiro para o próximo nó da lista, possui campos para armazenar o par $(chave, valor)$ que se pretende armazenar.

A inclusão de uma chave $x$ pode ser realizada no final da lista encadeada que corresponde ao endereço-base $h(x)$. Dessa forma, ao se realizar uma busca, para assegurar que uma chave não está na lista será necessário percorrer todos os seus

nós.

Para se excluir $x$, igualmente, será necessário realizar uma busca na lista encadeada do endereço $h(x)$. Após localizada a chave, basta fazer com que o ponteiro que apontava para o nó de $x$ passe a apontar para o nó apontando por $x$. Na figura 2.2 há a inclusão de várias chaves (começando da esquerda para direita) utilizando o tratamento de colisões por encadeamento exterior.

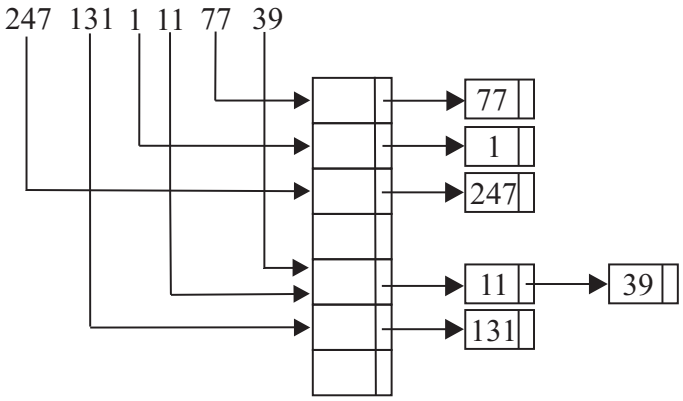


Figura 2.2: Encadeamento Exterior

Se a distribuição das chaves é uniforme, o custo médio de uma pesquisa nessa estrutura depende apenas do número médio de chaves nas listas encadeadas, ou seja, do fator de carga da tabela. O encadeamento exterior é eficaz mesmo quando o número de entradas na tabela é muito maior que seu tamanho. Seu pior desempenho ocorre quando todas as entradas são mapeadas para uma única posição, gerando uma lista encadeada de tamanho $n$. Logo, no pior caso, uma busca levaria $O(n)$ passos.

A eficiência desse método deve ser avaliada analisando-se quantas comparações são esperadas, em média, para buscas sem sucesso (seu pior caso). Então, suponha que $h$ é uma função uniforme, que o fator de carga da tabela é $\alpha$ e que as listas encadeadas utilizadas para solução das colisões são não ordenadas. Desta forma, a probabilidade de $h$ computar cada índice $k$ é uniforme e igual a $\frac{1}{m}$, e o número de comparações feitas ao se acessar a entrada $i$ da tabela é o comprimento esperado da

lista encadeada associada à posição $h(i)$. O que, de acordo com (SZWARCFITER; MARKENZON, 1994), é:

$$CM(T) = \frac{n}{m} = \alpha.$$

Já no caso da busca com sucesso, para saber quantas comparações são esperadas na média, imagine que para encontrar uma chave $x$, pesquisa-se uma lista de índice $k$. O número de comparações é proporcional ao comprimento da lista $k$ na posição em que $x$ foi inserida na tabela. Se $x$ foi a $j+1$-ésima chave incluída, então o comprimento médio da lista $k$ é $\frac{j}{m}$. Então, segundo (SZWARCFITER; MARKENZON, 1994), o custo médio esperado é dado por

$$CM(T) = 1 + \frac{\alpha}{2} - \frac{1}{2m}$$

.

Portanto, se o fator de carga é baixo (menor que uma constante $c$), a complexidade média da busca é $O(1)$.

#### *2.3.1.2 Encadeamento Interior*

O encadeamento interior é uma alternativa para casos onde não se deseja manter uma estrutura externa à tabela. Nesse método, os nós também possuem campos para armazenar o par $(chave, valor)$ e um ponteiro para a posição seguinte. Em caso de colisão, porém, o ponteiro indica posições na própria tabela de dispersão. Dessa forma, uma tabela com $m$ posições fica dividida em duas regiões de tamanho fixo. Uma reservada aos endereços-base com tamanho $p$, e outra reservada aos sinônimos, ou chaves colididas, com tamanho $s$, onde $s + p = m$.

Uma função de dispersão $h$ retorna apenas índices no intervalo $[0, ..., p-1]$ . O intervalo $[p, ..., m-1]$ é usado para armazenar as chaves colididas. Quando a área de colisões está completamente preenchida e há a inclusão de uma nova chave com colisão, ocorre o que é conhecido como falso *overflow*. Ou seja, pode haver posições vazias na área de endereçamento, mas como a área de colisão está cheia, a chave não pode ser inserida. Uma solução para esse problema, consiste no aumento do tamanho da área de colisão, acarretando a diminuição da área de endereços-base. No entanto, a eficiência também diminui, pois a área de endereçamento pode possuir no mínimo 1 posição e a área de colisão no máximo $m-1$, o que corresponde a uma lista encadeada, cujo tempo médio de busca é $O(n)$ (SZWARCFITER; MARKENZON, 1994).

A figura 2.3 mostra uma tabela de dispersão com encadeamento interior, onde a área de colisão está totalmente preenchida. Caso seja inserida uma nova chave com endereço-base igual a $0, 1$ ou $3$, ocorrerá uma colisão e essa chave deverá ser armazenada na área de colisão, mas como ela está cheia, ocorrerá um falso *overflow*.

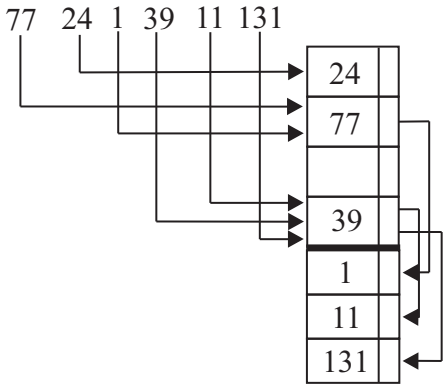


Figura 2.3: Tratamento de Colisões por Encadeamento Interior

Outra alternativa usando encadeamento interior consiste em destinar todo o espaço da tabela de dispersão para o endereçamento. Assim, qualquer posição da tabela pode ser usada como endereço-base ou para armazenar uma colisão.

Quando, na inserção de uma chave $x$, ocorre uma colisão, $x$ é armazenada na primeira posição livre após $h(x)$, digamos a posição $d$. Se, posteriormente, uma chave $y$ a ser incluída é tal que $h(y) = d$, será efetuada a fusão das listas correspondentes a $h(x)$ e $h(y)$, diminuindo assim a eficiência do método. Esta situação, que é o maior problema dessa abordagem, é chamada de colisão secundária, isto é, uma colisão entre chaves $x$ e $y$ para as quais $h(x) \neq h(y)$.

Na figura 2.4 a função de dispersão utiliza todo o espaço da tabela para endereçamento das chaves. Quando há colisão, a chave colidida é armazenada na primeira posição disponível de $m-1$ a 1, ou seja, na última posição da tabela que ainda está vazia. Por exemplo, a chave 49 foi mapeada para a posição 0, no entanto, a posição 0 já estava ocupada pela chave 77, então a chave 49 foi armazenada na última posição da tabela. Também é possível observar que houve uma colisão secundária entre as chaves 28 e 17, uma vez que a chave 28 foi mapeada para a posição 0 já preenchida. Ao percorrer a tabela de baixo para cima, foi encontrada a posição 3 para seu armazenamento, mas ao inserir a chave 17, cujo endereço $h(17) = 3$, a posição 3 já estava preenchida com a chave 28 e foi necessário procurar uma posição alternativa para armazenar 17, causando a fusão entre as listas de colisões das posições 0 e 3.

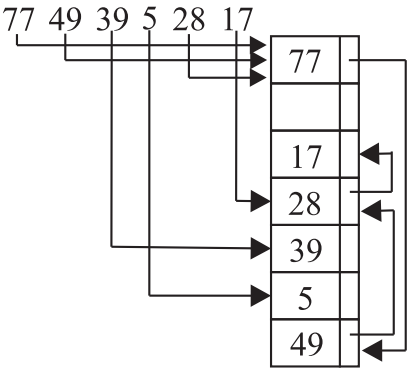


Figura 2.4: Encadeamento Interior sem Área de Colisão

No tratamento de colisões com encadeamento interior, seja com ou sem área

de colisão, há uma problema adicional que se refere às dificuldades introduzidas no processo de exclusão, uma vez que não se pode simplesmente retirar o elemento da lista. Para contornar essa situação considera-se que cada compartimento da tabela possa possuir três estados distintos: "vazio", quando o compartimento nunca foi ocupado; "ocupado", quando o compartimento contêm uma chave; e "liberado", que é o caso em que a solicitação de exclusão da chave foi realizada. Com isso, uma inserção posterior pode reaproveitar posições marcadas com "liberado", além de, quando for realizada uma busca, ser possível saber se há mais chaves na lista de colisão após o compartimento "liberado".

### 2.3.2 Tratamento de Colisões por Endereçamento Aberto

No endereçamento aberto todos os elementos são armazenados na própria tabela de dispersão, isto é, não existem listas nem elementos armazenados fora da tabela, evitando assim o uso de ponteiros. Em caso de colisão utiliza-se uma outra função de dispersão que indica o próximo índice a ser considerado para o armazenamento. Em geral, existe a função de dispersão $h(x, k)$, onde $x$ é a chave e $k \in \{0, ..., m-1\}$ é a sequência de tentativas, ou posições candidatas. Em uma busca pela chave $x$, calcula-se inicialmente $h(x, 0)$. Se a posição correspondente está ocupada, mas não pela chave desejada, então calcula-se $h(x, 1)$. Se ainda não for encontrada, mas a posição, igualmente, não se encontra vazia, então calcula-se $h(x, 2)$, e assim por diante. Com isso, a função de dispersão $h(x, k)$ deve ser determinada de forma que seja possível visitar todos os $m$ endereços em $m$ tentativas (KNUTH, 1998).

A busca com sucesso termina quando a posição que contêm a chave procurada é encontrada, e a busca sem sucesso é encerrada quando uma posição vazia é encontrada ou se esgotam todas as possibilidades.

É importante notar que, quando uma entrada da tabela é apagada, assim como no encadeamento interior, a posição não pode simplesmente ser de fato esvaziada, mas ser marcada como "liberada". Assim, suponha que existe a sequência de chaves $k_1$, $k_2$, $k_3$ armazenadas e todas foram mapeadas para a posição de $k_1$. Suponha que foi solicitada a exclusão da entrada $k_1$ e a posição $i$ que contêm a entrada $k_1$ tenha sido marcada como "vazia". A seguir, é realizada uma busca por $k_3$. Como a posição $i$ antes ocupada por $k_1$ está "vazio" a busca seria encerrada na posição $i$ e seria retornado um falso negativo. Ou seja, embora $k_2$ e $k_3$ estejam na tabela, por terem sido mapeados para $i$, que agora está "vazia", aquelas chaves não poderão ser encontradas! Por outro lado, se a posição $i$ estiver marcada como "liberada" (representando a deleção lógica do elemento que a ocupava), isto significa que essa posição pode ser utilizada para armazenar uma nova chave e, ao mesmo tempo, sinaliza para as buscas que ainda existem chaves mais adiante, permitindo encontrar $k_2$ e $k_3$ no exemplo dado.

Também é importante notar que, após um certo número de inserções e remoções, o desempenho das funções que utilizam a tabela de dispersão sofre uma queda considerável no desempenho, devido à falta de organização da estrutura.

No endereçamento aberto, pode-se perceber que o custo para buscar uma determinada chave é igual ao custo que foi necessário para inseri-la. Para uma função de dispersão de sequência uniforme, uma busca sem sucesso possui número médio de iterações igual a $\frac{1}{1-\alpha}$; já para uma busca com sucesso, onde as chaves possuem a mesma probabilidade de serem buscadas, assumindo que $0 < \alpha < 1$, o número médio de iterações é $\frac{1}{\alpha} log \frac{1}{1-\alpha} + \frac{1}{\alpha}$ (SZWARCFITER; MARKENZON, 1994).

A seguir são descritos alguns métodos para a determinação da sequência de tentativas $h(x,k), k = 0, ..., m-1$, para uma chave $x$ dada.

#### *2.3.2.1 Tentativa Linear*

Nesse método, ao calcular o endereço-base $h'(x)$ de uma chave $x$, se esse endereço já estiver ocupado por outra chave $y$ qualquer, tenta-se armazenar $x$ na posição $h'(x) + 1$, onde $h'$ denota um endereço-base obtido pelo uso de uma função de dispersão apresentada na seção 2.1. Se essa posição já estiver ocupada, tenta-se na posição $h'(x)+2$ e assim por diante, até encontrar uma posição vazia (SZWARCFITER; MARKENZON, 1994).

Então, a função de dispersão é

$$h(x, k) = (h'(x) + k) \bmod m, 0 \leq k \leq m - 1.$$

Contudo, a tentativa linear gera um problema decorrente da forma como trata as colisões. Esse problema é chamado de agrupamento primário e ocorre quando há a inclusão de várias chaves em sequência, cujos endereços-base já estejam preenchidos por outras chaves. Com isso, essas chaves inseridas são armazenadas no final do agrupamento aumentando-o ainda mais.

A figura 2.5 mostra o resultado da inserção de chaves usando o método da tentativa linear.

#### *2.3.2.2 Tentativa Quadrática*

Esse método tem como objetivo contornar o problema do agrupamento primário do método anterior, com a obtenção de uma sequência de endereços distinta para endereços-base próximos. Para tanto, é utilizado como incremento na função $h(x, k)$ uma função quadrática de $k$ (ZIVIANI, 1999).

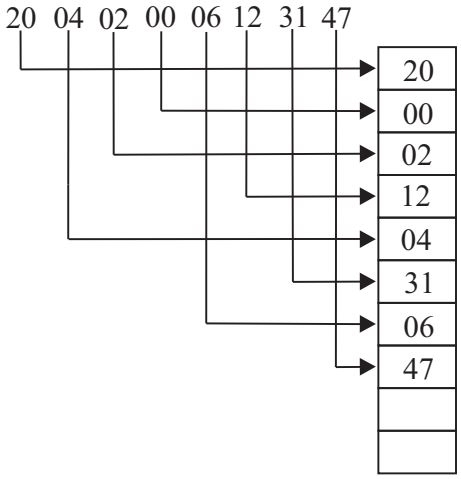


Figura 2.5: Tentativa Linear

Então a função de dispersão fica

$$h(x,k) = (h'(x) + c_1 k + c_2 k^2) \bmod m,$$

onde $c_1, c_2$ são constantes, $c_2 \neq 0$ e $k = 0, ..., m-1$ (SZWARCFITER; MARKENZON, 1994).

Pelo fato dos valores $m, c_1, c_2$ serem escolhidos, o desempenho do método depende dessa escolha. Ou seja, dependendo dos valores escolhidos, a função de dispersão pode não percorrer toda a tabela. Contudo, as equações recorrentes apresentadas por (SZWARCFITER; MARKENZON, 1994) fornecem uma maneira de calcular, diretamente, esses endereços.

$$h(x,0) = h'(x)$$

$h(x,k) = (h(x,k-1) + k) \bmod m$, $0 < k < m$ (SZWARCFITER; MARKENZON, 1994).

Apesar de resolver o problema do agrupamento primário, outro problema ocorre quando 2 chaves distintas possuem o mesmo endereço-base inicial, gerando outra forma de concentração de chaves na tabela, chamada de agrupamento secundário. Isso porque as sequências de tentativas das 2 chaves são idênticas. Então, a primeira chave é armazenada em uma posição definida pela sequência de tentativas, e a segunda chave só pode ser armazenada após o calculo de toda a sequência de endereços-base já calculados para a primeira. Apesar disso, a degeneração causada por esse agrupamento é menor que a do primário.

#### *2.3.2.3 Duplo Hash*

O duplo *hash* é uma opção que resolve os problemas de agrupamento primário e secundário, pois utiliza duas funções para gerar sua sequência de tentativas, conforme a função apresentada abaixo:

$$h(x,k) = (h'(x) + kh''(x)) \bmod m,\ 0 \leq k < m,$$

onde $h'$, $h''$ são funções de dispersão (SZWARCFITER; MARKENZON, 1994).

Os valores de $h''(x)$ e $m$ devem ser primos entre si, para que os endereços-base retornados pela equação acima corresponda a varrer toda a tabela (WIRTH, 1985).

Com esse método, a possibilidade da geração do mesmo endereço-base não é nula. Contudo, dadas duas chaves $x$ e $y$, se $h'(x) = h'(y)$ a probabilidade de $h''(x) = h''(y)$ é que na tentativa linear e na quadrática. Então, se uma boa função de dispersão for escolhida, o duplo *hash* elimina consideravelmente a possibilidade de que duas chaves que colidiram em uma determinada posição tenham a mesma sequência de tentativas (WIRTH, 1985).

Comparado aos demais métodos, o duplo *hash* tende a distribuir melhor as chaves na tabela de dispersão, já que, nas tentativas linear e quadrática, as sequências de tentativas obtidas eram idênticas se $h'(x) = h'(y)$, ao passo que no duplo *hash* as sequências só serão idênticas se, além de $h'(x) = h'(y)$, tivermos $h''(x) = h''(y)$.

## 2.4 Hashing Universal

Um fator crítico no projeto de um esquema de *hashing* está relacionado à escolha da função de dispersão. Prova-se que, para qualquer função de dispersão $h$, existe um conjunto $\lceil \frac{n}{m} \rceil$ de valores que são mapeados para a mesma posição da tabela de dispersão (TENENBAUM; LANGSAM; AUGENSTEIN, 1990). De modo que, dependendo da entrada, algumas funções de dispersão podem distribuir as chaves de maneira mais uniforme que as outras, sugerindo que, se diferentes funções de dispersão forem utilizadas, será obtido uma distribuição mais uniforme na média. Portanto, considere um conjunto de funções de dispersão cuidadosamente projetadas — priorizando, por exemplo, um baixo número de colisões para qualquer distribuição — como parte do esquema de *hashing*. Quando for necessário construir uma nova tabela de dispersão, será escolhida aleatoriamente uma função de dispersão que pertença a esse conjunto para ser usada na construção.

Assim, dado um conjunto $C$ de funções de dispersão que realiza o mapeamento de um universo $U$ de chaves no intervalo $\{0, 1, ..., m-1\}$, $C$ é chamado *hashing* universal se, para cada par de chaves distintas $x, y \in U$, o número de funções de dispersão $h \in C$ para a qual $h(x) = h(y)$ é igual a $\frac{|C|}{m}$, onde $m$ é o tamanho da tabela de dispersão (CARTER; WEGMAN, 1979).

Suponha que $C_{xy} = \{h : h(x) = h(y)\} : \dfrac{|C_{xy}|}{|C|} \leqslant \dfrac{1}{m}$. Então, se uma função aleatória $h$ for escolhida a partir de C, existe probabilidade menor ou igual a $\frac{1}{m}$ de

$h \in C_{xy}$, para quaisquer que sejam os dados.

Como exemplo de construção de um conjunto universal de funções de dispersão, considere as chaves que podem ser representadas como inteiros positivos no intervalo $\{0, 1, ..., M\}$, onde $M$ é o valor máximo de $U$. Suponha $p$ um número primo entre $M$ e $2M$ gerado de forma aleatória. Em seguida, defina para cada $b \in \{0, 1, ..., p-1\}$ e cada $d \in \{0, 1, ..., p-1\}$ a função

$$g_{b,d}(x) = bx + d(\text{mod } p).$$

Para cada $g$ defina uma função de dispersão

$$h_{b,d}(x) = g_{b,d}(x)(\text{mod } m).$$

O conjunto de todas estas funções $h_{b,d}(x)$ para determinados $M$ e $p$ é universal (CARTER; WEGMAN, 1979).

Se a tabela de dispersão não sofrer alterações durante as execuções (como o caso dos compiladores, por exemplo), a função de dispersão utilizada pode ser obtida a partir de um conjunto universal para assegurar um tempo médio de execução pequeno. Já se a tabela de dispersão sofrer alterações durante as execuções (como em bancos de dados, por exemplo), então é possível selecionar aleatoriamente uma função de dispersão inicial do conjunto universal, e se durante um número de execuções for detectado um comportamento deficiente da tabela de dispersão, é possível escolher aleatoriamente uma nova função de dispersão em cada execução, fazendo com que uma função de dispersão ruim seja trocada, mas exigindo que a cada mudança de função de dispersão a tabela inteira seja reorganizada.

## 2.5 Hashing Perfeito

O *hashing* universal garante que as operações básicas sejam realizadas em tempo médio $O(1)$, mas o pior caso continua sendo $O(n)$. Porém, quando o conjunto de entrada é estático (não sofre alterações), é possível encontrar uma função de dispersão que garante tempo $O(1)$, mesmo no pior caso (CORMEN et al., 2001).

Assim, dado um conjunto de chaves $K$, uma função de dispersão perfeita é uma função $h$ tal que $h(x) \neq h(y)$ para todo $x \neq y$.

Em geral, quanto maior a tabela de dispersão, mais fácil encontrar uma função de dispersão perfeita, mas é desejável que o tamanho da tabela de dispersão não seja muito maior que o tamanho do conjunto de entrada. Portanto, existe uma maneira simples de obter uma função de dispersão perfeita para tabelas de dispersão de tamanho $n^2$. E, mais recentemente, foi conseguido criar funções de dispersão perfeita para tabelas de tamanho $O(n)$ (CORMEN et al., 2001).

Quando a tabela de dispersão possui tamanho $n^2$, uma função de dispersão perfeita pode ser facilmente obtida por um algoritmo aleatorizado em tempo polinomial a partir de um conjunto *hashing* universal. Logo, para uma função aleatória de um conjunto *hashing* universal de funções de dispersão para uma tabela de dispersão de tamanho $m$, o número esperado de colisões é

$$E\left[\sum_{i \neq j} X_{ij}\right] = \sum_{i \neq j} E\left[X_{ij}\right] \geq \frac{n^2}{m}.$$

Portanto, caso uma tabela de dispersão de tamanho $m > n^2$ seja escolhida, o número esperado de colisões é menor que 1. Em particular, para $m > 2n^2$ a probabilidade de uma colisão é

$$P\left[\sum_{i \neq j} X_{ij} \geq 0\right] \leq \sum_{i \neq j} P\left[X_{ij} = 1\right] = \frac{n^2}{m} < \frac{1}{2} \text{ (CORMEN et al., 2001).}$$

O método de construção de *hashing* perfeito com espaço $O(n)$ é relativamente simples e elegante. O esquema possui dois níveis, utilizando *hashing* universal em cada nível. No primeiro nível, o mapeamento do conjunto de entrada é similar ao encadeamento, só que as chaves colididas são armazenadas em tabelas de dispersão secundárias, em vez de uma lista encadeada. Com isso, usando *hashing* universal, define-se uma função de dispersão correspondente a uma tabela de dispersão de tamanho $m$, onde $n = m$. Em seguida, todas as $n$ chaves são inseridas na tabela de dispersão, e as chaves colididas na posição $j$ armazenadas na tabela secundária $T_j$ correspondente.

Para garantir que não haverá colisões no segundo nível é necessário definir o tamanho de $T_j$ como sendo $n_j^2$, onde $n_j$ é o número de elementos $x$ cujo $h(x) = j$. Assim, no segundo nível é construída uma função de dispersão perfeita para cada $T_j$, de forma análoga à construção apresentada para tabelas com tamanho $n^2$. Apesar disso, foi provado que escolhendo bem a função de dispersão do primeiro nível, o espaço total esperado no esquema de dois níveis ainda é $O(n)$ (CORMEN et al., 2001).

A figura 2.6 ilustra a construção do *hashing* perfeito com espaço $O(n)$, onde é possível notar que dentre o conjunto de $n$ chaves, $n_j$ chaves foram mapeadas para posição $j$ pela função de dispersão do primeiro nível, e estas $n_j$ foram mapeadas pela função de dispersão daquela posição, ou seja, $h_j$ para a tabela de dispersão secundária $T_j$.

Quando um conjunto de $n$ chaves é mapeado para uma tabela de dispersão de tamanho $n$, sem colisões, diz-se que a função de dispersão perfeita é mínima (DIETZFELBINGER, 2007; TENENBAUM; LANGSAM; AUGENSTEIN, 1990). Ou seja, só é necessário um acesso para encontrar um elemento e não há posições vazias na tabela de dispersão.

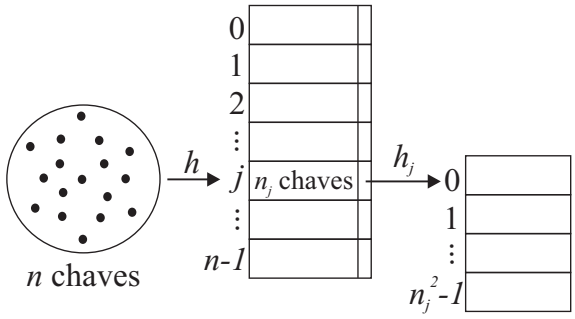


Figura 2.6: *Hashing* Perfeito com espaço $O(n)$

## 2.6 Hashing Dinâmico

A falta de flexibilidade no tamanho da tabela de dispersão e o alto custo para redimensionamento da mesma, tornam a utilização de *hashing* estático pouco atraente para armazenar conjuntos de entrada cujo tamanho varia no tempo. Para lidar com estes problemas, uma alternativa é a utilização de *hashing* dinâmico, que tem como objetivo fornecer uma função e uma tabela de dispersão adaptáveis às mudanças no tamanho do conjunto de entrada, mas que continue oferecendo desempenho equivalente ao do *hashing* estático.

Para tanto, no *hashing* dinâmico são alocados, inicialmente, uma tabela de dispersão de tamanho $m$ composta por $m$ compartimentos, onde cada compartimento é uma estrutura capaz de armazenar mais de um elemento. A maneira como o tamanho da tabela de dispersão varia é o que diferencia os métodos de *hashing* dinâmico, sendo o *hashing* linear uma opção bastante eficiente (SZWARCFITER; MARKENZON, 1994).

Considere uma tabela de dispersão de tamanho $m$. No *hashing* linear o espaço de endereços é aumentado aos poucos, expandindo compartimento a compartimento, começando do primeiro, e mantendo um ponteiro $pt$ para o próximo compartimento a ser expandido. Ao expandir um compartimento $p$, o compartimento $q$ expan-

dido de $p$, é inserido no final da tabela de dispersão e as chaves armazenadas em $p$ são distribuídas entre $p$ e $q$ de maneira conveniente. Após expandir todos os compartimentos, a tabela teve seu tamanho dobrado e o processo pode recomeçar, se necessário. A figura 2.7 apresenta um exemplo para $m = 3$, onde as setas indicam qual compartimento foi expandido e o resultante da expansão, o ponteiro para o próximo compartimento a ser expandido caso seja necessário e o pontilhado representa o local do compartimento expandido a partir do apontado por $pt$. Quando todos os compartimentos forem expandidos, a função de dispersão, neste caso, é alterada para $h(x) = x \bmod 6$.

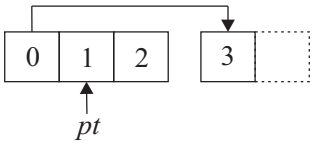


Figura 2.7: *Hashing* Linear

De maneira geral, a função de dispersão neste método pode ser definida como $h_l = x \bmod (m \cdot 2^l)$, onde $l$ é o número de vezes que o tamanho da tabela dobrou (SZWARCFITER; MARKENZON, 1994).

# 3 APLICAÇÕES

Este capítulo apresenta a solução de diversos problemas utilizando *hashing*. Para tornar o texto mais claro os problemas foram agrupados por similaridade.

## 3.1 Problemas: aritmético, algébrico e teoria de conjuntos

### 3.1.1 Números k-complementares

Seja $k$ um inteiro positivo, e seja $A$ uma lista não-ordenada contendo $|A| = n$ números inteiros. O problema dos números $k$-complementares consiste em determinar todos os pares não-ordenados $\{x, y\}$ de elementos de $A$ tais que a igualdade $x + y = k$ seja satisfeita. Seja, por exemplo, $A = [1, -4, 18, 11, 2, 9, -3, 5, 560, 10]$ e $k = 20$. A saída esperada compreende os pares $\{2, 18\}$ , $\{9, 11\}$ e $\{10, 10\}$.

Em uma primeira solução ingênua, cada elemento da lista $A$ é somado a cada um dos demais elementos, de forma que todos os pares de elementos de $A$ sejam investigados, e localizados aqueles que somem $k$. Procedendo desta forma, o problema certamente será resolvido, mas, embora nenhum espaço extra seja requerido,

a quantidade de adições que serão efetuadas será $O(n^2)$, resultando num algoritmo de tempo quadrático no tamanho da entrada.

Queremos fazer melhor. Primeiramente, algo interessante a ser observado é que, se selecionamos um elemento $x$ de $A$ e calculamos seu $k$-complemento $y = k - x$, o próximo passo consistirá em simplesmente procurarmos $y$, ou seja, nesse instante o problema passaria a ser um problema de busca. Gostaríamos, nesse caso, de poder buscar um elemento rapidamente.

Uma possibilidade seria utilizarmos vetores (ou *arrays*) booleanos para representarmos os elementos de $A$: cada elemento de $A$ indicaria uma posição preenchida com um *bit* 1, sendo todas as demais posições do vetor preenchidas com *bits* 0. Na técnica conhecida como endereçamento direto, a indicação da posição de cada elemento é conseguida fazendo com que o valor do elemento constitua seu próprio endereço, isto é, o índice de sua posição no vetor que o armazenará. Dessa forma, o elemento 18 ficaria armazenado na 18ª posição do vetor; o elemento 560, na 560ª posição; e assim por diante[1].

Há, no entanto, um problema grave nesta abordagem: o tamanho dos vetores deverá ser proporcional não ao tamanho de $A$, mas sim ao intervalo de valores possíveis que podem figurar em $A$, isto é, ao tamanho do universo de onde irão advir os valores da lista de entrada do problema. Se a lista contém, digamos, cem elementos positivos, o maior dos quais igual a um milhão, seria necessário um vetor com um milhão de posições para representar a lista! Mais formalmente, se $w$ é um parâmetro da entrada do problema, indicando o tamanho do universo de valores que poderão constar na lista, então o espaço utilizado pelo algoritmo seria exponencial no tamanho ocupado por este parâmetro na especificação da instância de entrada (uma vez que $w$ é representado por $\log w$ *bits*).

[1] O fato de poder haver números negativos em $A$ não chegaria a ser um problema, pois poderíamos usar dois vetores: um para os elementos não-negativos, outro para os elementos negativos de $A$, valendo-nos do módulo de cada número para a indicação de seu endereço no vetor correspondente.

Neste ponto nos remetemos ao emprego de *hashing*. Utilizando uma tabela de dispersão para armazenar os $n$ elementos de $A$, o espaço necessário será igual a $m = \frac{n}{\alpha}$. Para um fator de carga $\alpha$ constante, o tempo médio das operações de dicionário é também, como já mencionado, $O(1)$.

Já estamos diante de um algoritmo linear em espaço e em tempo médio para o problema dos números $k$-complementares. Percorre-se a lista $A$, armazenando seus elementos numa tabela de dispersão $H$ inicialmente vazia. A seguir, percorre-se $A$ uma segunda vez: para cada elemento $x$ de $A$, busca-se em $H$ o $k$-complemento $y$ de $x$, imprimindo $\{x, y\}$ caso $y$ esteja em $H$.

É claro que soluções utilizando *hashing* exigem uma elaboração um pouco maior na escrita do código do que soluções que utilizam apenas estruturas de dados mais simples como vetores e listas. Felizmente, a imensa maioria das linguagens de programação modernas de alto nível já oferecem – ou dispõem de bibliotecas que o fazem – implementações bastante eficientes de tabelas de dispersão, de forma que o programador não precisa necessariamente codificar as operações básicas para este tipo de estrutura, mas apenas utilizá-las, transparentemente, em seu proveito. De qualquer modo, para efeito de completude, apresentamos a seguir uma possível codificação do algoritmo proposto que dispensa qualquer suporte a *hashing* dado pela linguagem a ser utilizada.

Não é do escopo deste trabalho a concepção de funções de dispersão apropriadas a cada caso, assunto este que requer ferramental estatístico e estudo específico. A partir daqui, suporemos a existência de uma função de dispersão que mapeie valores $x$ de entrada em números inteiros do intervalo $[0, m-1]$, onde $m$ é o tamanho da tabela de dispersão utilizada[2]. Um exemplo possível seria a função muito simples $hash(x) = x \bmod m$ (KNUTH, 1998).

A solução proposta está dividida em três partes: um procedimento para a

[2]De fato, é possível encontrar boas funções de dispersão disponíveis nas modernas linguagens de programação.

criação e inicialização da estrutura básica, a tabela de dispersão; um procedimento para a inserção dos elementos de $A$ na tabela; e, por fim, o algoritmo propriamente dito, que soluciona o problema dos números $k$-complementares.

O procedimento *inicializa_hash()* cria uma tabela de dispersão com encadeamento exterior, na forma de um vetor $H$ com $m$ posições. Cada posição $i$ de $H$ contêm um ponteiro para a lista encadeada $L_i$, na qual serão armazenadas as chaves mapeadas na posição $i$ pela função de dispersão. As atribuições realizadas nas linhas $3, 4, 5$ e $6$ criam e inicializam as listas encadeadas associadas a cada posição de $H$. Cada uma destas atribuições é realizada em tempo $O(1)$, de forma que todo o procedimento roda em tempo $O(m)$.

**Algoritmo 1: Procedimento:** inicializa_hash($m$)

**Entrada**: inteiro $m$, indicando o tamanho desejado da tabela de dispersão
**Saída**: tabela de dispersão vazia
$H$ := novo vetor de tamanho $m$;
**para** *i := 0 até m-1* **faça**
    $L_i$ := nova lista;
    $L_i$.primeiro := $\emptyset$;
    $L_i$.ultimo := $\emptyset$;
    $H[i]$ := $_i$;
**fim**
retorne $H$

O procedimento *popula_hash( )* transfere os elementos da lista de entrada para uma tabela de dispersão previamente inicializada. Note que cada iteração do laço do procedimento *popula_hash( )* lê o próximo elemento da lista em tempo $O(1)$, calcula sua posição na tabela de dispersão também em tempo $O(1)$ e, por fim, armazena o elemento no endereço-base correspondente, o que é feito alocando espaço e atualizando ponteiros, cada uma dessas operações também executada em tempo $O(1)$. Logo, como o laço é repetido para todo elemento de $A$, a complexidade do procedimento é $n \cdot O(1) = O(n)$.

Finalmente, o algoritmo *busca_complementares( )* realiza a busca pelos pares

**Algoritmo 2: Procedimento:** popula_hash($A$,$H$)

**Entrada**: lista encadeada (ou vetor) $A$ contendo $n$ inteiros, tabela de dispersão $H$
**Saída**: $H$ contendo os elementos de $A$
**para** ***cada*** *elemento* $x$ *de* $A$ **faça**
    $h_x := hash(x)$;
    $L_h := H[h_x]$;
    $l :=$ novo item;
    $l$.chave $:= x$;
    $l$.proximo $:= \emptyset$;
    $l$.anterior $:= L_h$.ultimo;
    **se** $L_h.primeiro = \emptyset$ **então**
        $L_h$.primeiro $:= l$;
    **fim**
    $L_h$.ultimo $:= l$;
**fim**
retorne $H$

$\{x, y\}$ que somam $k$. Neste algoritmo, o pior caso ocorre quando todos os elementos de $A$ são mapeados para a mesma posição da tabela de dispersão, de forma que uma busca nesta tabela corresponda a uma busca numa lista encadeada de tamanho $n$. A complexidade de pior caso do algoritmo é, portanto, $n \cdot O(n) = O(n^2)$. No entanto, como se trata de uma solução com *hashing*, e como o espalhamento das chaves que é oferecida por boas funções de dispersão se aproxima de uma distribuição aleatória e uniforme (MORETTIN; BUSSAB, 2010), o pior caso deve ser visto não como uma entrada que leva o algoritmo a executar o maior número possível de passos, mas como uma função de dispersão que falha em distribuir os elementos de uma certa entrada de forma equilibrada. A probabilidade disto acontecer, no entanto, para qualquer entrada fixa, é em geral extremamente baixa. Em vista disto, a análise de caso médio, nos algoritmos envolvendo *hashing*, fornece um indicador bastante fiel de seu desempenho prático, independentemente da natureza das instâncias de entrada que lhe venham a ser fornecidas.

O segundo laço do algoritmo *busca_ complementares()* realizada a busca pelos

**Algoritmo 3:** busca_complementares($A$,$k$)

**Entrada**: lista encadeada $A$ contendo $n$ inteiros e um inteiro $k$
**Saída**: listagem de todo par $x, y$ tal que $x + y = k$

```
m = n/0.5 // arbitrando um fator de carga de 0.5;
H = inicializa_hash(m);
H = popula_hash(A,H);
para cada elemento x de A faça
    y := k − x;
    h_y := hash(y);
    L_h := H[h_y];
    pt := L_h.primeiro;
    enquanto pt ≠ ∅ faça
        se pt.chave = y então
            imprima x, y;
            pt := ∅;
        fim
        senão
            pt := pt.proximo;
        fim
    fim
fim
```

pares $\{x, y\}$. É possível apresentar duas análises de complexidade no caso médio: uma considera todos os conjuntos que possuem no mínimo um par $\{x, y\}$ que soma $k$. Essa análise irá retornar a complexidade de caso médio mínima, ou seja, a complexidade média para os melhores casos; a outra considera todos os conjuntos que não possuem elementos que somam $k$, de forma que, todas as pesquisas realizadas serão sem sucesso.

Inicialmente, suponha que exista pelo menos um par $\{x, y\}$, tal que $x+y = k$, e que a probabilidade de qualquer elemento ser armazenado em uma das $m$ posições seja igual. Além disso, suponha que o tempo para o endereço-base ser calculado seja $O(1)$.

O número de elementos examinados durante uma pesquisa com sucesso pelo

$y$, é uma unidade maior que o número de elementos que aparecem antes de $y$ em sua lista. É sabido que os elementos existentes antes de $y$ foram inseridos antes na tabela de dispersão, já que novos elementos são colocados no final da lista.

Então, seja $x_i$ o $i$-ésimo elemento adicionado a tabela. Em média, tem-se que o número de elementos $n_i$ que são mapeados para a mesma lista do elemento $x_i$ é

$$\sum_{(j=1)}^{n} \frac{1}{m} = \frac{1}{m} \sum_{(j=1)}^{n} 1 = \frac{1}{m}(n-1) = \frac{n-1}{m}.$$

Em média, o número de pesquisa necessária para encontrar o elemento $x_i$ é

$$1 + \sum_{(j=1)}^{n} \frac{1}{m} = 1 + \frac{n-1}{m}.$$

De onde tem-se que, em média, o número de pesquisas para encontrar algum elemento é a média de todos os $x_i$, o que pode ser calculado da seguinte forma, sabendo que $\frac{1}{n}$ é o número de comparações para encontrar o elemento $x_i$ para todo $x_i$:

$$\tfrac{1}{n}[\textstyle\sum_{i=1}^{n} 1 + \frac{(n-i)}{m}] = \tfrac{1}{n}[n + \sum_{i=1}^{n} \frac{(n-i)}{m}]$$

$$= 1 + \tfrac{1}{n}[\textstyle\sum_{i=1}^{n} \frac{(n-i)}{m}] = 1 + \frac{1}{nm}[\sum_{i=1}^{n}(n-i)]$$

$$= 1 + \tfrac{1}{nm}[\textstyle\sum_{j=0}^{n-1}(j)] = 1 + \frac{1}{nm}[\frac{n(n-1)}{2}]$$

$$= 1 + \tfrac{1}{m}[\tfrac{(n-1)}{2}] = 1 + \tfrac{n}{2m} - \tfrac{1}{2m}$$

$$= 1 + \tfrac{\alpha}{2} - \tfrac{\alpha}{2n}$$

Adicionando o tempo para calcular o endereço base temos $\Theta(2 + \frac{\alpha}{2} - \frac{\alpha}{2n})$. Então, somando a complexidade do primeiro e do segundo laço, tem-se que a complexidade no caso médio é $O(n) + \Theta(n-1) = O(n)$, para os conjuntos de entrada que possuem ao menos um par de números $k$-complementares.

Agora, considere apenas entradas onde não há qualquer par de números $k$-complementares, de forma que cada uma das buscas pelo $k$-complemento $y$ de um número $x$ da lista de entrada tenha que exaurir toda a lista encadeada correspondente ao endereço-base de $y$ na tabela de dispersão. Certamente, entradas que possuam pares de números $k$-complementares acarretarão certo número de buscas bem-sucedidas, buscas estas que serão concluídas possivelmente antes (mas nunca depois) que toda a lista correspondente tenha sido exaurida, de forma que o tempo gasto nestes casos será majorado pelo tempo gasto naqueles em que não há qualquer par de números $k$-complementares.

Sendo $T$ o tempo total para encontrar os $k$-complementos,

$$T = \sum_{i=1}^{n} T_i,$$

e pela linearidade da esperança,

$$E[T] = E[\sum_{i=1}^{n} T_i] = \sum_{i=1}^{n} E[T_i] = n \cdot O(1) = O(n).$$

Concluímos que, usando espaço linear no tamanho da entrada, o algoritmo proposto resolve o problema em tempo esperado linear, mesmo para entradas que não possuam qualquer par de números $k$-complementares. No algoritmo ingênuo analisado anteriormente, o tempo de pior caso era quadrático, e este limite é justo, por exemplo, nos casos de entradas sem pares $k$-complementares.

### 3.1.2 Interseção de conjuntos

O problema da interseção de conjuntos consiste exatamente no que seu nome indica: deseja-se encontrar elementos que pertençam simultaneamente a dois conjuntos $A$ e $B$. Há diversas formas de resolvê-lo, sendo a mais simples provavelmente aquela em que, para cada um dos elementos $x$ de um dos conjuntos, percorre-se possivelmente todo o outro conjunto à medida que seus elementos vão sendo comparados com $x$. Durante esse processo, imprimem-se os protagonistas de comparações bem-sucedidas. Sendo $n_A$ e $n_B$ os tamanhos de $A$ e $B$, respectivamente, são então realizadas $n_A \cdot n_B$ comparações no pior caso. Se fizermos $n$ igual ao tamanho do maior conjunto, entre $A$ e $B$, podemos descrever a complexidade de tempo de tal algoritmo como sendo $O(n^2)$.

Entretanto, é possível melhorar o algoritmo anterior ordenando-se previamente os elementos dos conjuntos e, em seguida, comparando-se os elementos dos dois conjuntos à medida que se caminha com dois índices, um para cada conjunto, como ilustrado pelo pseudocódigo do algoritmo *interseção_ ordenada( )*.

Os vetores $A$ e $B$ recebidos como entrada do algoritmo, são ordenados por qualquer algoritmo de ordenação, digamos *Merge Sort*, demandando para isto tempo $O(n \log n)$ no pior caso (CORMEN et al., 2001). Em seguida, a cada iteração do laço iniciado na linha 4, é efetuada a comparação entre um elemento de $A$ e outro de $B$, de forma que, se os elementos forem iguais, seu valor é impresso, e os índices $i$ e $j$ (de $A$ e $B$, respectivamente) são ambos incrementados em uma unidade. Se os elementos de $A$ e $B$ em questão forem diferentes, então, ou o índice $i$ é incrementado, caso o elemento de $A$ seja menor que o de $B$, ou em caso contrário é o índice $j$ que é incrementado. Assim sendo, o número de repetições do laço é igual ao número de incrementos realizados, que é $n_A + n_B$. Logo, a complexidade do algoritmo, se consideramos novamente $n$ como o maior dentre $n_A$ e $n_B$, é $O(n \log n)$.

Ainda é possível observar que, mesmo com a melhoria conseguida pela orde-

**Algoritmo 4:** interseção_ordenada($A$,$B$)

**Entrada**: vetores $A$ e $B$ contendo os elementos de dois conjuntos
**Saída**: $A \cap B$

```
Ordene A e B;
i := 0; j := 0;
n_A := |A|; n_B := |B|;
enquanto i < n_A e j < n_B faça
    se A[i]=B[j] então
        imprima A[i];
        i := i + 1; j := j + 1;
    fim
    senão
        se A[i]<B[j] então
            i = i + 1;
        fim
        senão
            j = j + 1;
        fim
    fim
fim
```

nação prévia dos elementos, a complexidade do algoritmo obtido ainda é maior do que o custo necessário para simplesmente percorrer todos os elementos de $A$ e $B$ uma única vez. Este fato nos leva a pensar em uma nova solução utilizando *hashing*.

A ideia é criar uma tabela de dispersão $H$ e populá-la usando os elementos de um dos vetores de entrada (digamos o menor dos vetores, para economizarmos espaço, e chamemo-no de $A$, sem perda de generalidade). A seguir, os elementos de $A \cap B$ são obtidos percorrendo-se todas as posições do vetor $B$ e comparando-se cada elemento $x$ de $B$ com os elementos $y$ que estão armazenados em $H$ no mesmo endereço-base que teria $x$, isto é, comparamos $x$ e $y$ quando $hash(x) = hash(y)$, onde $hash()$ é a função utilizada para popular $H$ (que ora omitimos pelos motivos já expostos, mas que pode ser tão simples quanto aquela apresentada no Capítulo 2).

O algoritmo *interseção_hashing( )* é dado a seguir em pseudocódigo, e o tratamento de colisões é feito por encadeamento exterior.

**Algoritmo 5:** interseção_hashing($A$,$B$)

**Entrada**: vetores $A$ e $B$, com $|A| \leq |B|$
**Saída**: $A \cap B$
$n_A := |A|$; $n_B := |B|$;
$m := \frac{n_A}{0.5}$ // arbitrando um fator de carga de 0.5;
$H =$*inicializa_hash*($m$);
$H =$*popula_hash*($A$,$H$);
**para** $i := 0, \ldots, n_B - 1$ **faça**
    $x := B[i]$;
    $h_x := hash(x)$;
    $L_x := H[h_x]$;
    $pt := L_x$.primeiro;
    **enquanto** $pt \neq \emptyset$ **faça**
        **se** $pt.chave = x$ **então**
            imprime $x$;
            $pt := \emptyset$;
        **fim**
        **senão**
            $pt := pt$.proximo;
        **fim**
    **fim**
**fim**

Os procedimentos *inicializa_hash( )* e *popula_hash( )* correspondem aqui aos procedimentos homônimos da Seção 3.1.1.

Uma análise análoga às que fizemos na Seção 3.1.1 nos mostra que o número médio de operações efetuadas pelo algoritmo é igual à quantidade de elementos do maior conjunto vezes $(1+\alpha)$, onde $\alpha$ é o fator de carga da tabela de dispersão. Como, ao utilizarmos espaço proporcional à quantidade de elementos a serem armazenados, estabelecemos $\alpha$ como uma constante, temos que a quantidade de comparações do algoritmo *interseção_hashing( )* é $O(n_B) \cdot \alpha = O(n_B)$. Esta quantidade de comparações, cada uma das quais realizada em tempo constante, somada às $O(n_A)$ operações para se popular a tabela com os elementos do menor conjunto, resulta numa com-

plexidade total esperada de $O(n_A) + O(n_B) = O(n)$ para o algoritmo. Trata-se, portanto, de um ganho considerável em relação aos tempos respectivos $O(n^2)$ e $O(n \log n)$ dos algoritmos anteriores.

### 3.1.3 Igualdade da soma de funções

O problema da igualdade da soma de funções consiste em encontrar as quádruplas ordenadas $(x, y, z, w)$ formadas por elementos de certo conjunto $A$ dado, satisfazendo $f(x) + f(y) = f(z) + f(w)$, para uma função real $f$ qualquer avaliada em tempo constante. Este problema pode ser considerado um caso particular e polinomial do problema da partição, que é NP-completo (HAYES, 2002), e no qual deseja-se saber se um conjunto de entrada $S$ pode ser dividido em dois conjuntos $S_1$ e $S_2$ de mesma soma, não havendo, contudo qualquer restrição ao tamanho dos dois conjuntos.

A partir deste ponto, não nos preocuparemos mais com a inicialização das tabelas de dispersão, ou com as funções de dispersão, ou os métodos de tratamento de colisão utilizados, que suporemos disponíveis e apropriados, citando-os apenas quando for conveniente.

A abordagem trivial do problema exposto examinaria cada uma das quádruplas de elementos do conjunto $A$, para uma complexidade de tempo igual a $O(n^4)$, onde $n = |A|$. Como transformar este problema num problema de busca em que possamos nos valer da eficiência média das tabelas de dispersão?

Uma resposta possível seria: armazenando os possíveis valores $d = f(x) + f(y)$, para todo $x, y \in A$, como chaves de uma tabela de dispersão $H$, cada uma das quais associadas a uma lista que conterá todos os pares ordenados $(x, y)$ cujas imagens por $f$ somem $d$. A seguir, para cada chave $d$ armazenada em $H$, combinamos dois a dois os pares $(x, y)$ pertencentes à lista associada a $d$, obtendo assim as

quádruplas desejadas. O pseudocódigo do algoritmo *soma_de_funções_hashing( )* é dado a seguir:

**Algoritmo 6:** soma_de_funções_hashing($A$,$f$)

```
Entrada: vetor A, função f
Saída: quádruplas ordenadas (x, y, z, w) satisfazendo
       f(x) + f(y) = f(z) + f(w)
H = tabela de dispersão inicialmente vazia;
para cada (x, y) ⊆ A faça
    d = f(x) + f(y);
    se d é chave de H então
        L_d := lista encadeada associada a d, em H;
    fim
    senão
        L_d := nova lista encadeada;
        insira em H a chave d e, associada a ela, a lista L_d;
    fim
    acrescente o par (x, y) à lista L_d;
fim
para cada d em H faça
    L_d := lista encadeada associada a d, em H;
    para cada (x, y) ∈ L_d faça
        para cada (z, w) ∈ L_d faça
            imprima (x, y, z, w);
        fim
    fim
fim
```

O espaço utilizado será aquele necessário para armazenar $O(n^2)$ pares numa tabela de dispersão, isto é, $\Theta(n^2)$, para um fator de carga constante.

Quanto ao tempo, cada chave $d$ pode ser localizada e/ou inserida em $H$ em tempo médio constante, e cada um dos $O(n^2)$ pares de elementos $(x, y)$ de $A$ demanda tempo $O(1)$ para ser inserido no final da lista associada à chave $d = f(x) + f(y)$. Dessa forma, toda a tarefa de se popular a tabela com cada chave $d$ associada aos pares de elementos cujas imagens somem $d$ pode ser executada em tempo médio $O(n^2)$.

Resta listar as quádruplas que satisfazem a igualdade desejada (linhas 10-14), o que é conseguido pelas diferentes combinações dos pares armazenados em uma mesma lista. Como é gasto tempo constante para cada par impresso, e sendo $d$ a quantidade de tais pares, o tempo esperado de todo o algoritmo será $O(n^2 + d)$.

## 3.2 Algoritmo de ordenação - *Bucket Sort*

O *Bucket Sort* é um algoritmo de ordenação por distribuição que consiste na definição de intervalos nos quais o conjunto de entrada será dividido. Os elementos de cada intervalo são armazenados em um *bucket*, e cada *bucket* corresponde a somente um intervalo. Os valores depositados em cada *bucket* são ordenados utilizando outro algoritmo de ordenação, ou o próprio *Bucket Sort* recursivamente. Ao final, o conjunto ordenado é obtido percorrendo os *buckets* em ordem (CORMEN et al., 2001).

É uma ideia muito simples e pode facilmente ser implementada criando um vetor de tamanho $m$, onde $m$ é o maior elemento do conjunto de entrada e, portanto, o último na lista ordenada. Cada elemento é armazenado na posição do vetor cujo índice é igual ao seu valor (endereçamento direto). Após realizada a distribuição, percorre-se o vetor a partir da posição de menor índice para obter o conjunto ordenado (CORWIN; LOGAR, 2004).

Nesta implementação, como cada posição pode armazenar apenas um valor, haverá perda de informação quando utilizada na ordenação de conjuntos que possuem elementos repetidos. Com isso, foi proposta uma alternativa na qual cada *bucket* armazena quantas vezes o elemento cujo valor igual aquele índice aparece no conjunto de entrada, conforme mostra a figura 3.1.

No que se refere a complexidade, nas duas implementações é necessário percorrer os $n$ elementos do conjunto de entrada realizando a distribuição das informações

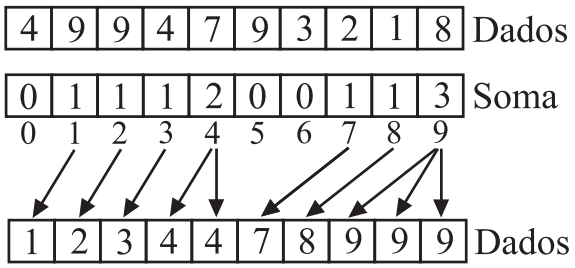


Figura 3.1: *Bucket Sort*

entre as $m$ posições do vetor; e no final, percorrer as $m$ posições do vetor, para obter o conjunto ordenado, ou seja, a complexidade é $O(n + m)$. Contudo, é necessário criar um vetor tão grande quanto o maior elemento do conjunto possa ser, sendo que, na maioria dos casos, poucas posições serão utilizadas. Assim, há uma troca entre tempo e espaço, pois estas implementações possuem complexidade de tempo inferior a qualquer algoritmo de ordenação por comparação e troca, cujo o limite inferior é $O(n \log n)$, mas exige uma demanda muito maior de espaço.

Para continuar obtendo bom desempenho sem a necessidade de uma grande quantidade de espaço, uma alternativa para a implementação do *Bucket Sort* é a utilização de tabelas de dispersão, onde cada posição da tabela representa um *bucket*. Neste caso, a definição dos intervalos e o número de elementos que cada *bucket* espera receber, depende do tamanho da tabela de dispersão, da qualidade da função de dispersão e do fator de carga. Além disso, considere dois elementos $x$ e $y$, e suponha sem perda de generalidade $x < y$. A função de dispersão deve assegurar que $x$ seja mapeado para uma posição da tabela igual ou menor que a posição de $y$.

É importante observar que a definição da função de dispersão depende do conjunto de entrada. Se a entrada for uniformemente distribuída, os elementos são distribuídos de maneira uniforme pela tabela de dispersão definindo uma função de dispersão que crie *buckets* de tamanhos iguais (intervalos com o mesmo tamanho), mas se a distribuição dos elementos da entrada não for uniforme, ainda é possível

definir uma função de dispersão que distribua os elementos de maneira uniforme pela tabela. Para tanto, é necessário que sejam criados intervalos menores na faixa de valores em que os elementos estão mais concentrados, e ir aumentando o intervalo à medida que os elementos possuem menor probabilidade de estarem presentes no conjunto de entrada.

Considere o caso em que a entrada é uniformemente distribuída. Seja $max$ o maior elemento do conjunto e $m$ o tamanho da tabela de dispersão, de modo que $m \leq n$. O intervalo armazenado em cada *bucket* é igual a

$$i = \frac{max}{m}$$

e a função de dispersão pode ser definida como

$$h(x) = \left\lfloor \frac{x}{i} \right\rfloor .$$

O algoritmo *bucketsort_hashing()* mostra a implementação do *Bucket Sort* utilizando a função de dispersão definida anteriormente. Observe que o ponto crítico com relação a complexidade do algoritmo está no primeiro laço, onde os elementos são incluídos na tabela de dispersão, realizando a ordenação por inserção em cada *bucket*.

Então, para definir o tempo esperado do algoritmo *bucketsort_hashing()* devemos contar o número de elementos que foram armazenandos e ordenados em cada *bucket* $i$. Logo, seja $X_i$ uma variável aleatória binomial que conta os elementos armazenados no *bucket* $i$ e $T$, o tempo total gasto pelo algoritmo. Temos

$$T = \sum_{i=1}^{m} X_i^2,$$

onde $X_i^2$ se deve ao tempo de execução quadrático do algoritmo de ordenação por inserção utilizado na ordenação dos *buckets*.

**Algoritmo 7:** bucketsort_hashing($A, m, max$)

**Entrada**: $A$: vetor a ser ordenado; $m$: tamanho da tabela de dispersão; e $max$: maior valor de $A$
**Saída**: tabela de dispersão com os valores ordenados

```
n := |A|;
i := max/m;
para j := 0 até n-1 faça
    x := A[j];
    h_x := ⌊x/i⌋;
    L_h := H[h_x];
    pt := L_h.primeiro;
    enquanto pt ≠ ∅ faça
        se pt. chave < x então
            pt := pt.proximo;
        fim
        senão
            l := novo item;
            l.chave := x;
            l.proximo := pt;
            l.anterior := pt.anterior;
            pt := ∅;
        fim
    fim
fim
para j := 0 até m faça
    pt := L_j.primeiro;
    enquanto pt ≠ ∅ faça
        imprime pt.chave;
        pt := pt.proximo;
    fim
fim
```

Tomando a esperança em ambos os lados, temos

$$E[T] = E[\sum_{i=1}^{m} X_i^2].$$

Sabe-se que a variância de uma variável aleatória binomial é $np(1-p)$ (onde

$n$ é o número de tentativas e $p$ é a probabilidade de sucesso (MORETTIN; BUSSAB, 2010)) e que $E[X]^2 = (n \cdot p)^2$. Então,

$$Var(X) = E[X^2] - E[X]^2$$

$$np(1-p) = E[X^2] - (np)^2$$

$$E[X^2] = np - np^2 + n^2p^2$$

Fazendo $p = \frac{1}{m}$ temos

$$E[X^2] = \frac{n}{m} - \frac{n}{m^2} + \frac{n^2}{m^2}$$

$$E[X^2] = \frac{n}{\Theta(n)} - \frac{n}{(n^2)} + \frac{n^2}{\Theta(n^2)}$$

$$E[X^2] = \Theta(1) - \frac{1}{\Theta(n)} + \Theta(1)$$

$$E[X^2] = \Theta(1) - \frac{1}{\Theta(n)}$$

Substituindo,

$$T = \sum_{i=1}^{m} \left\{ \Theta(1) - \frac{1}{\Theta(n)} \right\}$$

$$T = \sum_{i=1}^{m} \Theta(1) - \sum_{i=1}^{m} \frac{1}{\Theta(n)}$$

$$T = \Theta(m) - \frac{m}{\Theta(n)} = \Theta(m) = \Theta(n)$$

Logo, independente da complexidade do algoritmo de ordenação utilizado em cada *bucket*, a complexidade esperada do *Bucket Sort* é $\Theta(n)$. Mesmo nos casos em que a entrada não é uniformemente distribuída, o *Bucket Sort* ainda pode ser executado em tempo linear (CORMEN et al., 2001).

## 3.3 Algoritmos com leitura/escrita de dados em tempo médio constante

### 3.3.1 Matrizes esparsas

Uma matriz é dita esparsa quando apresenta uma grande quantidade de posições com valor igual a zero (ZIVIANI, 1999). São utilizadas nas mais diversas áreas, como, por exemplo, em otimização, meteorologia, análise de redes elétricas, simulação de circuitos, bioinformática, e na computação para representar grafos e planilhas eletrônicas, entre outras.

As operações usuais realizadas sobre matrizes (soma, multiplicação, inversão, pivotamento) não utilizam as posições com valor igual a zero, o que permite, no caso das matrizes esparsas, economizar uma quantidade significativa de espaço e realizar as operações em tempo muito menor, armazenando somente os valores diferentes de zero. Fato esse, que tem levado ao desenvolvimento de diversas formas de armazenamento comprimido, que consideram desde características gerais da matriz, até características mais específicas, como, por exemplo, se possuem blocos de valores diferentes de zero. Em geral, os formatos desenvolvidos podem ser utilizados para armazenamento de qualquer matriz, mas as melhorias em termos de desempenho e espaço são enfatizadas se utilizados para armazenar as esparsas.

Existem na literatura diversos formatos de representação de matrizes esparsas por conjunto de vetores (ZIVIANI, 1999; BAI et al., 2000). Sendo os formatos Compressed Row Storage (CRS) e Compressed Column Storage (CCS) os mais gerais, uma vez que não fazem nenhum tipo de suposição com relação a distribuição dos elementos diferentes de zero, e por isso, serão aqui descritos.

O formato CRS, originalmente sugerido por D. J. Rose (GALLAGHER, 1972), coloca os valores diferentes de zero, que são subsequentes na matriz esparsa em posições contínuas da memória. Então, sendo $A$ uma matriz esparsa, cria-se 3 vetores para armazená-la: o vetor $Val$ armazena os valores da matriz que são diferentes de zero; o vetor $ColInd$ armazena os índices das colunas em que se encontram os valores armazenados em $Val$. Ou seja, se $Val(k) = a_{ij}$, então $ColInd = j$; e, por fim, o vetor $RowPtr$ armazena os índices de $ColInd$ em que o vetor $Val$ inicia uma nova linha, isto é, se $Val(k) = a_{ij}$, então $RowPtr(i) \leq k \leq RowPtr(i+1)$. Assim, sendo $nz$ o número de elementos diferentes de zero da matriz esparsa, são necessários $2nz + n + 1$ locais para armazenar a matriz $A$ (BAI et al., 2000). Uma economia de espaço significativa, uma vez que se toda a matriz esparsa fosse armazenada teria um total de $n \times m$, ou $n^2$ elementos armazenados.

Considere a matriz esparsa $A$ apresentada na figura 3.2:

$$A = \begin{bmatrix} 7 & 0 & 0 & 0 & 1 & 0 \\ 0 & 9 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & -8 & 0 & 0 & 3 \\ 0 & 0 & 0 & 9 & 0 & 0 \\ 0 & 15 & 0 & 0 & -4 & 0 \end{bmatrix}$$

Figura 3.2: Matriz Esparsa $A$

A tabela 3.1 mostra a representação da matriz da figura 3.2 utilizando o formato CRS. Além dos valores $Val$, $ColInd$ e $RowPtr$ foram apresentados os índices dos vetores.

| k | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | 7 | 1 | 9 | 5 | 6 | -8 | 3 | 9 | 15 | -4 |
| ColInd | 1 | 5 | 2 | 4 | 6 | 3 | 6 | 4 | 2 | 5 |

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| RowPtr(i) | 1 | 3 | 5 | 6 | 8 | 9 |

Tabela 3.1: Representação da matriz $A$ utilizando CRS

O formato de representação CCS é análogo ao CRS. Ou seja, $ColInd$ é equivalente a $RowInd$, logo para $Val(a_{ij})$, $RowInd = i$ e $ColPtr$ é equivalente a $RowPtr$, então $ColPtr$ será o índice do elemento $Val$ em $RowInd$, onde começa uma nova coluna de $A$.

Apesar dos formatos CRS e CCS utilizarem menos espaço que o armazenamento tradicional e serem muito utilizados, não são os mais eficientes para realizar operações matemáticas em matrizes, porque recorrem a um método de endereçamento indireto para acessar cada valor.

Neste cenário, outra opção de representação de matrizes esparsas faz uso de uma lista duplamente encadeada. Permitindo que os elementos sejam percorridos tanto por linha quanto por coluna, facilitando as operações sobre as matrizes. Cada elemento é armazenado em um nó da lista, que será uma estrutura composta pelo índice da coluna, índice da linha, o valor do elemento, um ponteiro para o próximo elemento que está na mesma linha e um ponteiro para o próximo elemento que está na mesma coluna (ZIVIANI, 1999).

Na representação por listas encadeadas, somente os valores diferentes de zero da matriz esparsa são armazenados. Cada coluna e cada linha da matriz são representadas por uma lista circular com um nó cabeça, ou seja, cada nó pertence a duas listas simultaneamente. Então, um nó pode ser alcançado a partir do nó cabeça que corresponde a linha de sua posição na matriz, ou a partir do nó cabeça que correspondem a coluna de sua posição.

Considere a matriz esparsa $B$ apresentada na figura 3.3:

$$B = \begin{bmatrix} 8 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

Figura 3.3: Matriz Esparsa $B$

Uma possível representação da matriz $B$ usando listas duplamente encadeadas pode ser vista na figura 3.4, onde os nós cabeça podem ser identificados pelo valor $-1$ nos campos linha e coluna.

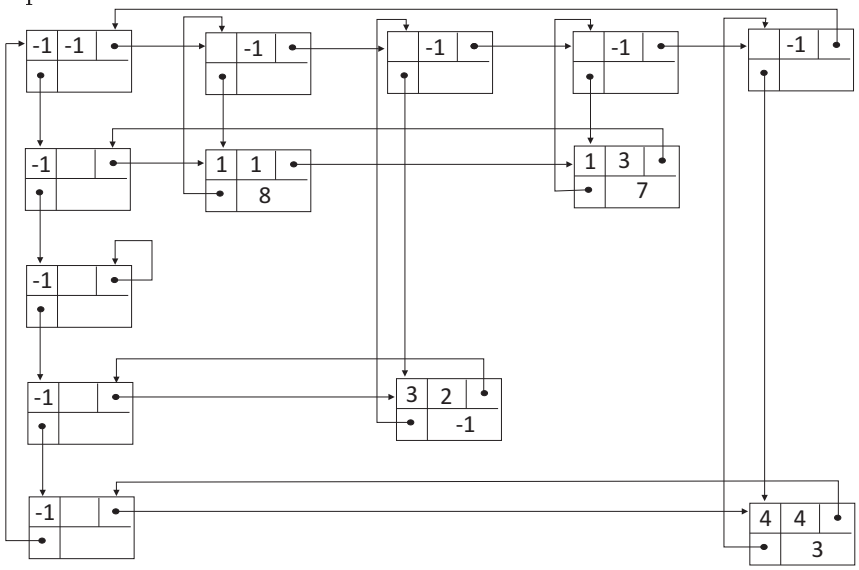


Figura 3.4: Representação da matriz $B$ utilizando lista encadeada

A representação de uma matriz esparsa $n \times m$ com $r$ elementos diferentes de zero, usando listas encadeadas, gastará $(m + n + r)$ nós (ZIVIANI, 1999). Apesar de um nó ocupar vários *bytes* de memória, o total de memória usado — quando o $r$ for suficientemente pequeno — ainda será menor que as $n^2$ posições de memória necessárias para representar toda a matriz.

Um problema existente neste tipo de representação está relacionado ao fato de não ser possível acessar um determinado índice diretamente, sendo necessário

varrer a estrutura até o índice desejado. Além disso, os formatos CRS e CCS ocupam menos espaço que a representação por listas encadeadas, embora seja necessário mais tempos para realizar modificações na matriz. Essas questões sugerem a proposta de uma nova abordagem, que consiste, basicamente, num formato de armazenamento de coordenadas, onde todos os valores diferentes de zero são armazenados em uma tabela de dispersão (ASPNAS; SIGNELL; WESTERHOLM, 2007).

Então, seja $A$ uma matriz esparsa $n \times m$. Considere um elemento $(i, j)$ com valor $v$, onde $i \in [0, n-1]$, $j \in [0, m-1]$. Para definir a posição de $v$ na tabela de dispersão, é necessário definir uma chave para $v$, de forma que seja possível efetuar seu endereçamento na tabela e recuperar, posteriormente, sua posição na matriz. Como são conhecidos apenas $v$ e sua posição na matriz, pode-se definir uma chave $k$, como sendo, por exemplo, $k = i * n_{col} + j$, onde $i$ é a linha e $j$ é a coluna de $v$ na matriz. O tamanho da tabela de dispersão pode ser da ordem do número de elementos diferentes de zero na matriz, pois assumindo que os elementos diferentes de zero estão uniformemente distribuídos sobre as chaves definidas, tem-se que, em média, o número de elementos mapeados para cada posição da tabela de dispersão utilizando encadeamento exterior será $\alpha$ (fator de carga).

A figura 3.5 mostra a representação da matriz $A$ da figura 3.2 utilizando *hashing* com encadeamento exterior, onde foi utilizada a função de dispersão $h(x) = k \bmod 5$ e a chave $k = i^2 + j$, onde $i$ é a linha e $j$ é a coluna de $x$ em $A$.

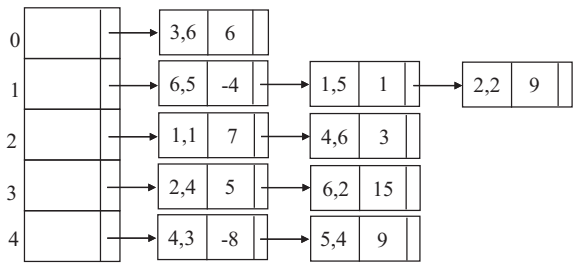


Figura 3.5: Representação da matriz $A$ utilizando *hashing*

A inclusão de um valor $x$ na posição $(i, j)$ da matriz é realizada calculando a

chave $k$, e armazenando a coordenada $(i, j)$ e $x$ na posição $h(k)$. Cada posição da tabela de dispersão é uma estrutura composta pela coordenada $(i, j)$ e um valor $x$, o que torna possível a identificação de coordenadas distintas em uma mesma lista de colisão. Não é necessário armazenar a chave $k$, uma vez que pode facilmente ser obtida realizando uma operação básica em tempo $O(1)$.

Para realizar buscas, é necessário calcular a chave $k$ e o endereço-base $h(k)$, e depois percorrer a lista encadeada verificando se a chave procurada encontra-se na lista. Na exclusão, efetua-se uma operação de busca, e se o valor for encontrado, a remoção é realizada fazendo a atualização dos ponteiros do antecessor e sucessor do valor, e liberando o espaço ocupado pelo nó.

Esta representação de matrizes esparsas permite que as dimensões da matriz sejam facilmente alteradas, porém não permite a realização de uma ordenação dos valores. As buscas são rápidas, uma vez que cada posição da tabela de dispersão terá uma lista encadeada com comprimento médio $\alpha$, embora no pior caso a complexidade seja $O(n)$. Uma abordagem alternativa seria a utilização de uma estrutura baseada em árvore para tratar as colisões, por exemplo, uma árvore AVL (GOODRICH; TAMASSIA, 2001). Isto faria com que as buscas pudessem ser realizadas em tempo $O(\log n)$, mas as inclusões seriam mais complexas e seria necessário o uso de mais espaço para armazenar os dois ponteiros para as subárvores esquerda e direita, respectivamente, além dos campos para a posição $(i, j)$ e o valor.

### 3.3.2 Representação de Grafos

Um grafo é uma estrutura definida como $G = (V, E)$, onde $V$ é um conjunto não vazio de elementos denominados vértices, e $E$ é o conjunto de pares $(v_i, v_j)$ denominados arestas, onde $v_i, v_j \in V$. Esta estrutura pode ser representada computacionalmente de diversas formas (SZWARCFITER, 1984), dentre as quais podemos

destacar as representações unívocas por matriz de adjacência e por listas de adjacência.

Dado um grafo $G = (V, E)$, na representação por matriz de adjacência, é definida uma matriz $A$ de tamanho $n \times m$, de forma que $a_{ij} = 1$ se os vértices $v_i$, $v_j$ forem adjacentes e $a_{ij} = 0$, caso contrário. Já na representação por listas de adjacência, habitualmente é utilizada uma estrutura mista, constituída por um vetor $V$ que representa os vértices do grafo. Sendo cada posição de $V$ associada a uma lista encadeada, na qual são armazenados os vértices adjacentes ao vértice em questão.

Devido a grande similaridade entre a estrutura usualmente utilizada para representar um grafo por listas de adjacência e a tabela de dispersão com encadeamento externo, é intuitivo utilizar *hashing* para tal representação (TENENBAUM; LANGSAM; AUGENSTEIN, 1990). À vista disso, ao utilizar *hashing* para representar as listas de adjacência de um grafo, cada nó é mapeado para uma posição de uma tabela de dispersão e seus vizinhos são mapeadas para uma tabela de dispersão associada a tal posição. Observe que assim como na representação usual, cada aresta $(v, w)$ dá origem a duas entradas na tabela de dispersão — uma corresponde a exploração da vizinhança de $v$ e outra explorando a vizinhança de $w$. Desta forma, esta representação utiliza um espaço igual a $O(m \times \alpha)$, onde $m$ é o tamanho da tabela de dispersão na vertical, ou seja, aquela para a qual os vértices são mapeados, e $\alpha$ é tamanho das tabelas onde é armazenada a vizinhança de cada vértice. Como $\alpha$ é constante, o espaço é linear no tamanho de $G$. A figura 3.6(b) mostra a representação do grafo da figura 3.6(a) utilizando *hashing*.

Nesta representação, o tempo para executar uma busca por uma aresta $(v, w)$ é, na média, $O(1)$ para encontrar o vértice $v$ na tabela de dispersão vertical, mais $O(1)$ para encontrar o vértice $w$ na tabela de dispersão associada a posição de $v$, enquanto que na representação utilizando listas encadeadas associada a um vetor,

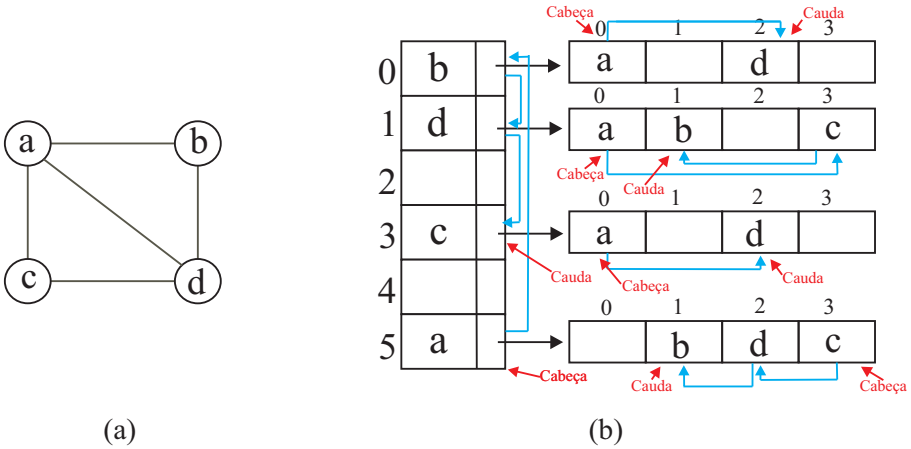


Figura 3.6: Representação de grafos usando *hashing*

para realizar a mesma busca seria necessário percorrer o vetor até encontrar o vértice $v$ e depois percorrer uma lista encadeada associada, até encontrar $w$, ou o final da lista, caso os vértices não sejam adjacentes.

Note que a tabela de dispersão da vertical e as tabelas associadas a cada uma de suas posições são encadeadas, ou seja, possuem ponteiros que indicam a sequência em que os vértices foram inseridos. Esses ponteiros são extremamente úteis ao realizar a listagem das listas de adjacências do grafo, pois permitem que o número de posições visitadas nas tabelas de dispersão seja exatamente o mesmo que na representação usual. Isto é, dado que se deseja listar as listas de adjacência do grafo, basta tomar o nó cabeça da lista encadeada da tabela da vertical e para cada nó nesta tabela, tomar o nó cabeça da tabela associada ao nó em questão (tabela da horizontal) e percorrer esta lista até o nó cauda. Dessa forma, somente as posições não vazias das tabelas de dispersão serão visitadas.

É interessante observar que matriz de adjacência é esparsa, e como visto na seção 3.3.1, tal representação também pode ser realizada utilizando *hashing*, reduzindo o espaço utilizado de $O(n^2)$, para um número na ordem da quantidade de 1's existentes na matriz.

## 3.4 Tabelas de dispersão encadeadas

### 3.4.1 Cache LRU

O cache é um método bastante conhecido por melhorar a eficiência e escalabilidade de pesquisas realizadas em meios de armazenamento que possuem acesso lento. Basicamente, mantêm as informações mais relevantes — sob um determinado ponto de vista — em locais que possuem acesso de alta velocidade.

Um dos aspectos importantes na implementação de um cache, está ligado à política de substituição, que define quais informações serão removidas quando o cache estiver cheio. Um algoritmo de substituição bastante eficiente, com alta taxa de acerto e baixa complexidade é o LRU (Least Recently Used). A ideia por traz desse algoritmo é que informações acessadas recentemente têm maior chance de serem utilizadas nos próximos acessos e, portanto, devem permanecer armazenadas. Logo, quando o cache não possuir espaço livre para inserir novas informações, com o LRU, as informações que estão mais tempo sem serem utilizadas são removidas.

Uma possível implementação do cache LRU pode ser feita através de uma lista duplamente encadeada, mantendo os elementos mais recentemente utilizados no início da lista e os menos recentemente usados no final (TANENBAUM, 2001). Quando o cache está cheio e é necessário inserir um novo elemento, libera-se espaço, excluindo o elemento final da lista, que é o menos recentemente utilizado. Portanto, nas substituições do cache, retira-se o elemento que está no final da lista e insere o novo no começo. Com o encadeamento duplo, em cada acesso a um elemento existente no cache, é possível movê-lo para o começo da lista, tornando-o o novo elemento mais recentemente utilizado. Realizar essa movimentação e efetuar pesquisas nessa implementação possuem custo elevado. Além disso, esta é uma boa abordagem para caches com acessos únicos, pois nos casos em que vários meios podem acessá-lo

simultaneamente, sempre que uma movimentação, exclusão e inclusão de um nó é realizada, o cache precisa ser bloqueado, causando lentidão.

O cache LRU também pode ser implementado utilizando uma lista duplamente encadeada interna à tabela de dispersão, chamada lista LRU. Cada posição da tabela de dispersão, que também é um nó da lista LRU, é uma estrutura composta pelo par $(chave, valor)$. Nesta implementação, a inserção é realizada com o mapeamento da chave para uma posição da tabela de dispersão, e para preservar a ordem dos acessos, o nó inserido é adicionado como nó cabeça de uma lista LRU. Em uma pesquisa com sucesso, a chave buscada é movimentada para a primeira posição da lista LRU, através da atualização de ponteiros. Quando é realizada uma busca sem sucesso e o cache está cheio, torna-se necessário abrir espaço para o novo elemento ser inserido. Tal operação pode ser facilmente realizada, excluindo o último nó da lista LRU, que também é o elemento menos recentemente utilizado. Na figura 3.7 é mantido o cache LRU de páginas *web* acessadas. Observe que as páginas foram acessadas na sequência $\text{URL}_1$, $\text{URL}_2$, $\text{URL}_3$, $\text{URL}_8$, $\text{URL}_5$, $\text{URL}_2$ e que no segundo acesso a $\text{URL}_2$ houve atualização de ponteiros na lista LRU.

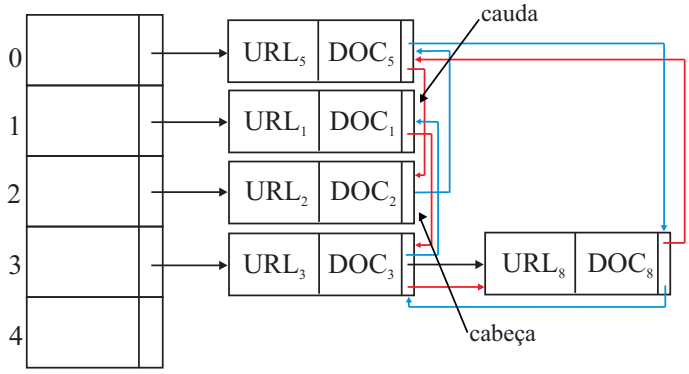


Figura 3.7: Cache LRU com Tabela de dispersão e Lista Duplamente Encadeada

Quanto a complexidade, temos que toda chave é mapeada para uma posição da tabela de dispersão e o nó é inserido na lista LRU, permitindo que o tempo médio das buscas seja $O(1)$, para um fator de carga $\alpha$ constante. Além disso, o nó

correspondente ao elemento mais recentemente utilizado está sempre no topo da lista e o menos recentemente utilizado está sempre no final, permitindo que inclusões e remoções sejam realizadas em tempo constante. A complexidade de espaço é O($m$), onde $m$ é a capacidade máxima do cache.

Entretanto, a utilização da tabela de dispersão com lista duplamente encadeada não resolve o problema de lentidão causado pelos bloqueios, principalmente em caches que precisam permitir acessos múltiplos.

Recentemente, (SPYCHER, 2011) apresentou uma variação dessa implementação que além da tabela de dispersão e da lista duplamente encadeada, utiliza uma lista para armazenar os elementos que devem ser excluídos. Neste caso, as inserções também são realizadas através do mapeamento das chaves entre as posições da tabela de dispersão. Quando uma chave é acessada, são criados: um nó associado a essa chave na lista LRU; e um ponteiro entre o elemento na tabela de dispersão e na lista LRU, caso ainda não exista. Dessa forma, os nós da lista LRU ficam externos à tabela de dispersão, e são mantidos de forma que, os mais recentemente utilizados ficam no final da lista e os menos recentemente utilizados no começo. Além da inserção, são admitidas as operações de busca, movimentação do nó entre as posições da lista LRU, remoção dos nós e exclusão dos nós marcados na lista de exclusão.

Quando uma busca é realizada, se o elemento for encontrado, o nó é movido para o final da lista LRU, indicando que esse elemento foi acessado recentemente. Em uma sequência de buscas e movimentação de nós, não há bloqueio, então, quando as operações de busca são concorrentes, não pode-se dizer que o nó movido para a última posição da lista LRU corresponde ao elemento mais recentemente acessado, mas sim um dos mais recentemente acessados.

Após cada inserção, se o cache atingiu um determinado tamanho próximo a sua capacidade, o nó cabeça da lista, que é um dos menos recentemente acessados, é adicionado à lista de nós a serwm excluídos e o ponteiro que o associa a uma entrada

da tabela de dispersão é definido como vazio, é como se fossem criados buracos no cache. Quando a lista de nós a serem excluídos atingir um tamanho pré-definido, os nós nela armazenados são de fato excluídos da lista LRU. Essa operação de exclusão é limitada, porque envolve duas atualizações (um ponteiro ao sucessor e outro para o nó antecessor), além de ser necessário bloquear o cache, para garantir uma exclusão segura e correta.

Com o uso da lista para armazenar os nós que devem ser excluídos, eliminou-se a obrigatoriedade de excluir um nó toda vez que o cache estiver cheio, permitindo que a exclusão seja realizada em lotes, reduzindo o número de vezes em que é necessário bloquear o cache. Mas seu uso exige mais espaço que as implementações anteriores.

### 3.4.2 Problema do Mercado Financeiro

Certa empresa $X$ atua no mercado financeiro negociando papéis com um grande número de parceiros. Diariamente, diversas negociações são feitas, muitas das quais de forma automática por softwares especializados. Ao longo de um dia típico, um parceiro pode realizar novas negociações com $X$ ou cancelar as negociações realizadas até aquele instante, pois as negociações do dia só são de fato efetivadas após o horário comercial. Não há limites para as quantidades de negociações ou de cancelamentos.

Cada negociação feita por $X$ recebe um código de portfólio, que, para todos os efeitos, é um número inteiro associado àquela operação. Pelas regras de negócio de $X$, no entanto, operações com parceiros distintos podem vir a receber o mesmo código, e operações com um mesmo parceiro podem receber códigos distintos.

As operações efetuadas ao longo de um dia são armazenadas, uma por linha, em um arquivo. Cada linha compreende um sinal de "+" ou de "-". Um sinal de "+" indica que uma nova negociação foi feita, e é sempre seguido do identificador

de um parceiro comercial de $X$ e do código de portfólio. Um sinal de "-", por outro lado, é seguido apenas do identificador de um parceiro comercial de $X$ e indica o cancelamento de todas as negociações realizadas com aquele parceiro, desde o início do dia até o momento em que aquela entrada foi registrada no arquivo, não importando seus códigos de portfólio. Tanto os identificadores de parceiros quanto os de portfólio são números inteiros advindos de um intervalo muito grande.

A tabela 3.4.2 mostra a estrutura básica desse arquivo de negociações.

| | | |
|---|---|---|
| + | 101 | 145113 |
| + | 2 | 26 |
| + | 101 | 26 |
| + | 3005 | 4550 |
| - | 101 | |
| + | 3005 | 145113 |
| + | 4 | 184 |
| + | 101 | 4550 |
| - | 2 | |

Tabela 3.2: Estrutura do arquivo de negociações

Diariamente, após o encerramento das negociações, a empresa precisa detectar quais portfólios estão associados a uma ou mais negociações ativas, isto é, que não foram canceladas. Para obter essa informação é necessário percorrer o arquivo de operações. Mas qual a melhor maneira de fazê-lo?

É fácil pensar em diversas maneiras de processar o arquivo de operações para obter a informação desejada em tempo quadrático (ou pior) no tamanho do arquivo. Por exemplo, percorrendo-se o arquivo na ordem cronológica das entradas (isto é, de cima para baixo), podemos verificar para cada linha indicando parceiro $Y$ e portfólio $P$, se houve cancelamento das operações de $Y$ em qualquer uma das linhas subsequentes, imprimindo $P$ caso não tenha havido (e possivelmente cuidando para que o mesmo portfólio não seja impresso mais de uma vez). Examinaremos, agora, soluções mais eficientes utilizando *hashing*.

Considere a leitura do arquivo ainda sendo feita na ordem em que as negociações foram realizadas, ou seja, da primeira linha à última linha, cronologicamente. Utilizaremos duas tabelas de dispersão: uma, a de parceiros por portfólio, onde as chaves serão os códigos de portfólio, e os valores armazenados serão listas contendo os parceiros de $X$ que têm negociações ativas sob aquele código de portfólio; e outra, a de portfólios por parceiro, onde as chaves serão os parceiros de $X$, e os valores associados a cada chave serão listas contendo os códigos de portfólio utilizados em negociações ativas com aquele parceiro.

Cada linha lida do arquivo com o sinal "+" indicando parceiro $Y$ e portfólio $P$ acarretará duas alterações nas estruturas de dados utilizadas: a primeira é a inclusão de $Y$ na lista de parceiros associados à chave $P$ na tabela de dispersão de parceiros por portfólio (a chave $P$ será inserida pela primeira vez, caso ainda não se encontre na tabela); a segunda alteração é a inserção de $P$ na lista de portfólios associados à chave $Y$ na tabela de portfólios por parceiro (a chave $Y$ será inserida pela primeira vez, caso ainda não se encontre na tabela).

Quando uma linha com o sinal "-" é lida para um parceiro $Y$, é preciso excluir as ocorrências de $Y$ de ambas as tabelas. Na tabela de portfólios por parceiro, $Y$ é chave e, pode, portanto, ser recuperada e removida em tempo médio $O(1)$. Na tabela de parceiros por portfólio, no entanto, $Y$ pode pertencer a listas associadas a vários portfólios. Para não ser necessário percorrer todas as listas, usamos a informação da tabela de portfólios por parceiro (antes da exclusão da chave $Y$, evidentemente), para já saber, de antemão, quais são os portfólios em cujas listas $Y$ estará na tabela de parceiros por portfólio. Poderemos, assim, ir diretamente naquelas listas e remover $Y$, sem necessidade de percorrer toda a tabela.

Ao fim do processamento de todas as linhas, basta retornar os portfólios que estiverem associados a listas não-vazias de parceiros, na tabela de parceiros por portfólio.

A Figura 3.8 ilustra as estruturas de dados utilizadas nesta primeira solução proposta. Observe que, por clareza, não aparecem na figura chaves distintas com o mesmo endereço-base. Como mencionado anteriormente, deixamos o tratamento de colisões a cargo da implementação das tabelas de dispersão utilizadas, e focamos no emprego da técnica.

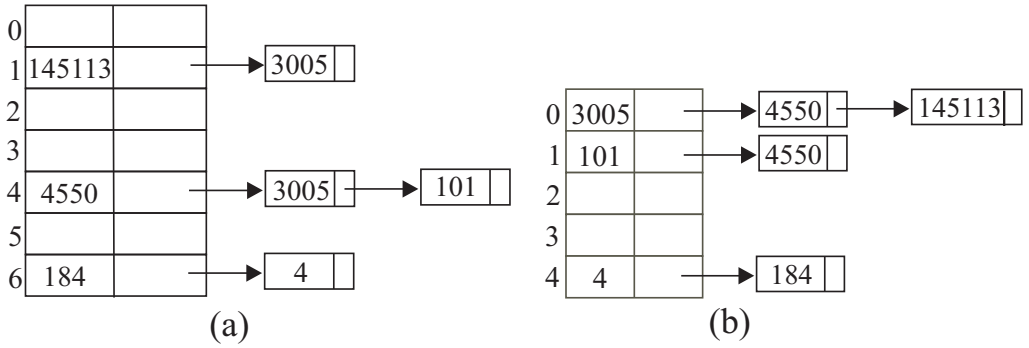


Figura 3.8: (a) parceiros por portfólio: tabela de dispersão com listas (b) portfólios por parceiro: tabela de dispersão com listas

Como vimos, a localização de cada lista de parceiros da qual $Y$ precisa ser removido pode ser feita em tempo médio $O(1)$, pois trata-se da busca por uma chave (o código do portfólio) em uma tabela de dispersão. No entanto, para localizar $Y$ dentro de cada uma das listas, precisaríamos percorrer toda a lista, o que certamente aumentaria o tempo computacional. Não surpreende, porém, a esta altura, que possamos substituir com vantagem as listas contendo os parceiros associados a cada portfólio por uma tabela de dispersão contendo tais parceiros! Em outras palavras, o valor associado à cada chave $P$ na tabela de dispersão original (parceiros por portfólio) será o ponteiro para uma tabela de dispersão cujas chaves serão os parceiros que negociaram com a empresa $X$ sob o código de portfólio $P$ (veja a Figura 3.9). Sendo assim, diante de uma operação de cancelamento, a localização e a remoção de cada parceiro $Y$ será feita também em tempo médio $O(1)$, e todo o algoritmo rodará então em tempo esperado linear no número de linhas do arquivo.

A solução mais elegante para o problema do mercado financeiro, no entanto,

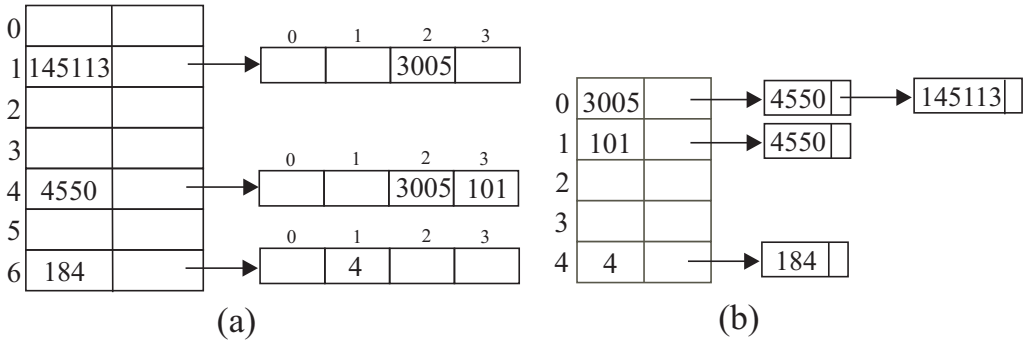


Figura 3.9: (a) parceiros por portfólio: tabelas de dispersão aninhadas (b) portfólios por parceiro: tabela de dispersão com listas

consiste na leitura do arquivo de operações de trás para frente, isto é, da última para a primeira negociação realizada no dia, evitando, como mostraremos a seguir, o processamento de negociações que viriam a ser, de todo modo, posteriormente canceladas.

Note que, como o arquivo estará sendo lido de trás para frente, então a primeira linha encontrada com o sinal de "-" indicando o cancelamento das operações passadas com um certo parceiro $Y$ é referente, na verdade, ao último cancelamento (cronologicamente) das operações com $Y$. Dessa forma, essa linha irá particionar todas as demais linhas referentes à $Y$ em dois grupos: as que ocorreram cronologicamente antes desse cancelamento (e que não foram ainda processadas, pois se encontram acima dela, no arquivo), e as que ocorreram cronologicamente após este cancelamento (e que, portanto, já foram processadas uma vez que se encontram abaixo dela no arquivo). A partir de então, as linhas do primeiro grupo poderão ser totalmente ignoradas, uma vez que foram, de todo modo, canceladas. Por outro lado, as linhas que já foram processadas não terão sido canceladas, por terem ocorrido após o último cancelamento envolvendo $Y$!

Usaremos novamente duas tabelas de dispersão: uma para os portfólios ativos, que serão retornados no final; outra para indicar parceiros comerciais "cancelados", isto é, parceiros para os quais já foi processada uma linha indicando o ultimo can-

celamento de suas operações naquele dia. Com essa estrutura, cada linha com o sinal "-" verá o parceiro comercial em questão ser adicionado à tabela de parceiros cancelados (cujas operações serão ignoradas a partir de então); e cada linha com o sinal "+", se referente a um parceiro que ainda não foi cancelado, verá o portfólio em questão ser adicionado à tabela de portfólios ativos (de onde jamais sairá!); caso contrário, será simplesmente ignorada.

A Figura 3.10 mostra como as negociações da Tabela 3.4.2 são armazenadas durante a execução do algoritmo proposto.

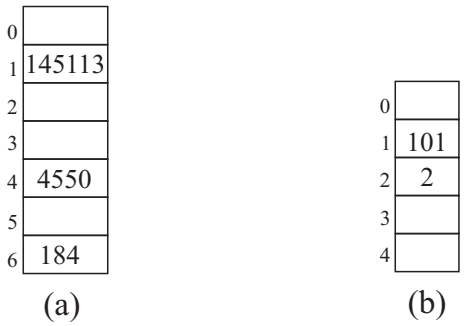


Figura 3.10: (a) portfólios ativos: tabela de dispersão apenas com chaves (b) parceiros cancelados: tabela de dispersão apenas com chaves

Como tanto a busca de um parceiro na tabela de parceiros cancelados quanto a inserção de um portfólio na tabela de portfólios ativos podem ser realizadas em tempo médio $O(1)$, todo o processo demanda tempo esperado linear no tamanho do arquivo. Além disso, contando-se o número de parceiros cancelados e comparando-se esse número com o total de parceiros com os quais a empresa negociou ao longo do dia (informação esta que pode estar disponível), torna-se dispensável a leitura do restante do arquivo, a partir do momento em que esses dois números se tornem iguais, uma vez que as demais linhas só irão conter negociações posteriormente canceladas, e o algoritmo pode parar antes mesmo que todas as linhas do arquivo tenham sido processadas.

## 3.5 Busca Exaustiva (*Backtracking*)

Quando inexistem algoritmos eficientes para tal, a busca pela solução ótima de um problema de otimização combinatória é em geral implementada como uma busca exaustiva em um grafo implícito, cujos nós correspondem a soluções viáveis — ou estados, ou configurações válidas — do problema em questão. Suas arestas correspondem a escolhas possíveis numa sequência de decisões que permite obter todos os estados ou configurações atingíveis a partir do ponto de partida, o estado inicial. Em outras palavras, uma aresta $xy$ indica que existe um passo, uma tomada de decisão na sequência pré-estabelecida, que nos leva diretamente da configuração ou estado $x$ à configuração ou estado $y$. O que se deseja é encontrar um nó do grafo que maximiza ou minimiza determinada função, ou que constitui um "estado-alvo", atendendo certa condição de parada. Técnica conhecida como *backtracking*.

Ao realizar uma busca exaustiva é necessário ter atenção quanto à sequência de configurações, pois, se não forem tomados os devidos cuidados, configurações iguais, simétricas ou equivalentes podem ser visitadas mais de uma vez, acarretando perda de tempo ou laços intermináveis. Por esta razão, é preciso demarcar, ao longo da busca, os nós do grafo já visitados. O que nos leva invariavelmente a um problema de espaço, pois o conjunto de todas as configurações possíveis — atingíveis ou não, mas como sabê-lo? — pode ser enorme, e por conseguinte a quantidade de memória alocada para a desejada demarcação.

Diante dessas questões, é possível propor uma abordagem que promove uma redução considerável de tempo e espaço, como ilustrado na solução dos problemas definidos a seguir.

### 3.5.1 Carga da Balsa

Como se sabe, balsas são embarcações usadas para transportar veículos através de cursos d'água. Normalmente possuem largura suficiente para formar duas pistas com veículos de diferentes tamanhos, ocupando todo o seu comprimento. Os veículos aguardam pela embarcação em uma fila única, onde sua ordem na fila determina sua ordem de embarque. O problema Carga da Balsa consiste em determinar como carregar a balsa com o maior número possível de veículos, onde cada veículo pode ocupar a pista da direita ou a da esquerda. Se não houver espaço livre para o veículo à frente da fila, a balsa está cheia e os veículos restantes devem aguardar a próxima balsa (SKIENA; REVILLA, 2003).

Para compreender a dificuldade em encontrar uma solução para o problema, considere o problema do escalonamento de tarefas com duas máquinas idênticas. Este problema é NP-completo (GAREY; JOHNSON, 1979). Agora considere a fila de veículos esperando para serem embarcados na balsa. Some o comprimento dos primeiros veículos da fila até que o comprimento total obtido seja igual a $2k$ (onde $k$ é o comprimento da balsa) ou o mais próximo possível. Esta subfila de soma $\leq 2k$ é a entrada do problema do escalonamento, onde cada veículo é uma tarefa a ser alocada e seu comprimento corresponde ao tempo de execução. Então, neste momento, o problema consiste em definir se existe uma alocação das tarefas de forma que o tempo gasto por cada máquina seja $\leq k$.

Se a resposta for sim, então os veículos da subfila podem ser embarcados na balsa, e para obter a ordem em que devem ser carregados, basta tomar uma solução do problema do escalonamento e permutar os veículos de cada lado da balsa, de modo a respeitar a ordem inicial da fila. Já se a resposta for não, quer dizer que mesmo com a ordem folgada não é possível carregar todos os veículos da subfila. Logo, se uma ordem for fixada também não será possível carregá-los.

Isto mostra que encontrar uma solução para o problema Carga da Balsa é equivalente a encontrar uma solução para o problema do escalonamento com duas máquinas idênticas. Logo, o problema carga da balsa é NP-completo.

Para se obter a solução do problema Carga da Balsa é necessário definir uma sequência de carregamento dos veículos na pista da esquerda e da direita, conforme mostrado na figura 3.11:

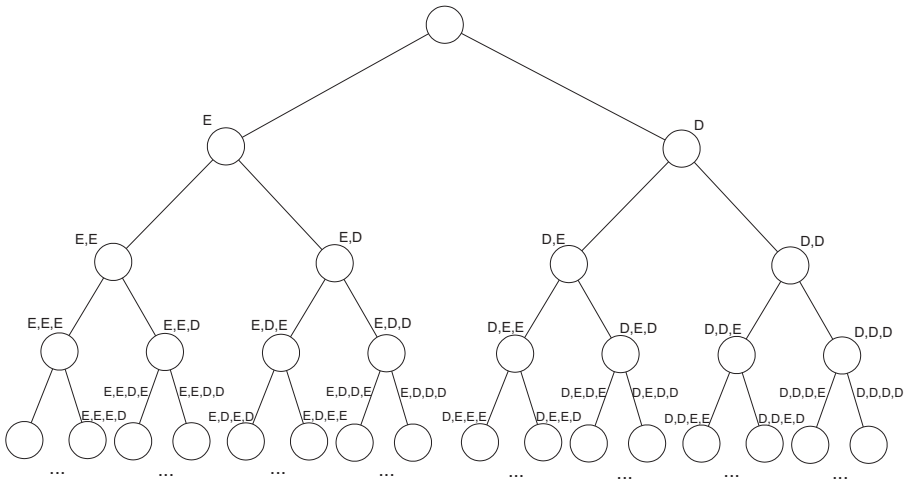


Figura 3.11: Árvore de decisão – Sequência de direções

Note que o número de nós nos níveis da árvore cresce de forma exponencial à medida que descemos por ela. Ou seja, o nível 0 tem um nó (a raiz), o nível 1 tem 2 (esquerda ou direita), o nível 2 tem 4 nós, e assim por diante. Generalizando, podemos dizer que no nível $d$, a árvore tem $2^d$ nós. Então, uma solução para o problema que utiliza essas configurações será exponencial no número de carros embarcados.

Uma alternativa para o problema Carga da Balsa, é considerar uma configuração válida como sendo definida pelas coordenadas $(x, y)$, onde $x$ é o espaço ocupado pelos veículos que foram embarcados na pista da esquerda e $y$, o espaço ocupado na pista da direita. A figura 3.12 mostra todas as configurações possíveis para um caso em que a balsa possui comprimento igual a 8 unidades de espaço e os veículos que

estão na fila para serem embarcados possuem comprimento igual a 2, 2, 4, 2, 4, 4, 5, 7.

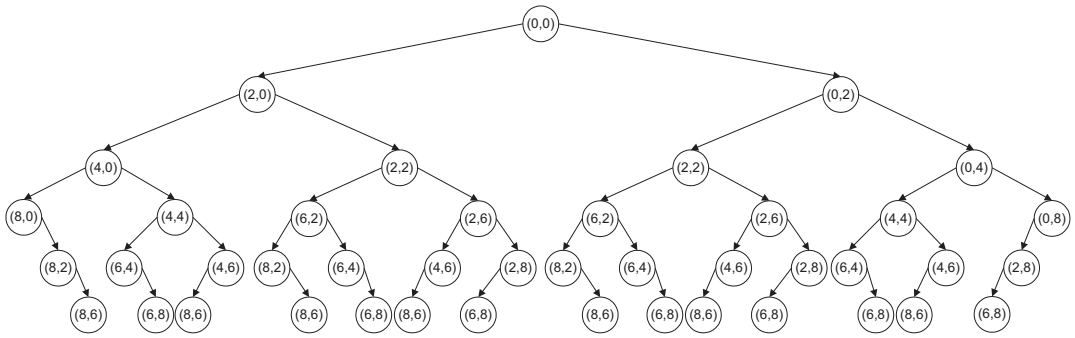


Figura 3.12: Árvore de decisão – Espaço ocupado

Observe na figura 3.12 que as melhores soluções ou as soluções ótimas ocupam as folhas da árvore, uma vez que a partir destas configurações, a balsa não possui espaço livre suficiente para embarcar o próximo veículo da fila, e, com isso, outra configuração não pode ser obtida. Então, podemos concluir que a melhor sequência de carregamento da balsa pode ser obtida "subindo" de uma folha $h$ até a raiz da árvore de decisão. Mas como saber quais configurações foram visitadas entre $h$ e a raiz?

Como o *backtracking* funciona como uma busca em um grafo implícito, dado que chegamos a uma folha $h$, certamente teremos passado pelos nós entre $h$ e a raiz da árvore. Além disso, podemos perceber que o lado esquerdo da árvore é simétrico ao lado direito, o que exige um cuidado maior ao armazenar as configurações visitadas, já que não é desejável que uma configuração seja visitada mais de uma vez. Então, para armazenar as configurações visitas e não permitir que uma mesma configuração seja visitada mais de uma vez, podemos utilizar uma tabela de dispersão, com tamanho proporcional ao número de nós visitado, na qual as chaves correspondem ao espaço ocupado a direita e a esquerda da balsa em cada configuração (considerando sempre que o maior espaço ocupado está à esquerda para facilitar a detecção de repetições). Quando uma nova configuração for visitada, verifica se já existe na

tabela uma configuração igual ou equivalente, e caso não exista, a nova configuração é armazenada, e adicionada em uma lista encadeada interna à tabela que mantêm a ordem em que as configurações foram visitas, para que quando a configuração alvo for atingida, ou seja, quando o número de veículos embarcados for o maior possível, seja possível recuperar a ordem em que foram carregados.

No problema Carga da Balsa, duas configurações $x$ e $y$ são equivalentes se, em $x$, os veículos embarcados ocupam exatamente a mesma porção de cada pista que é ocupada pelos veículos embarcados em $y$, a menos de simetria. Uma vez visitada uma dessas configurações, não é necessário visitar qualquer outra. Isto reduz o número de nós da árvore de busca, que seria exponencial na quantidade de veículos, para um número que é pseudo-polinomial no tamanho da balsa.

Entretanto, é possível definir uma solução para o problema Carga da Balsa por programação dinâmica, que utiliza menos espaço. Logo, o algoritmo proposto serve para ilustrar como seria a solução do problema utilizando *backtracking*.

### 3.5.2 Quebra-cabeça do 15

Desenvolvido na década de 1870 por Sam Loyd, o Quebra-cabeça do 15 é um jogo popular composto por uma grade $4 \times 4$ com quadrados numerados de 1 a 15 e um espaço vazio. O objetivo do jogo é a partir de uma ordem aleatória dos quadrados, obter sua ordenação crescente, de modo que o espaço vazio ocupe a última posição abaixo e a direita da grade (SKIENA; REVILLA, 2003; BALL, 1926).

O problema pode ser expandido para uma versão $n \times n$. Nesta formulação mais geral, o problema de encontrar a solução com o menor número de movimentos é NP-difícil (RATNER; WARMUTH, 1990). Na verdade, o problema de decidir se existe uma sequência de movimentos que dada uma configuração inicial, chega a configuração alvo em menos de $k$ movimentos é NP-completo.

Para alcançar a configuração final do jogo o único movimento válido é a troca de posições entre os quadrados numerados e o espaço vazio. Cada posição ocupada pelo espaço vazio permite que, no máximo, sejam realizados os movimentos de troca (deslizamento) "U", "D", "R" ou "L", ou seja, efetuar a troca do espaço vazio com o quadrado acima, abaixo, à direita ou à esquerda. A figura 3.13 mostra que partindo da configuração inicial apresentada, são necessários apenas 3 movimentos para chegar a configuração alvo. Observe que o número de movimentos possíveis varia entre 2 e 4, dependendo da posição ocupada pelo espaço vazio. Por exemplo, partindo da configuração inicial apresentada, se for realizado o movimento "D", a partir desta nova configuração só é possível realizar os movimentos "U", "R" e "L". Os pontos abaixo de cada configuração, indicam que é necessário realizar outros movimentos para obter a configuração alvo a partir da configuração em questão, caso seja possível.

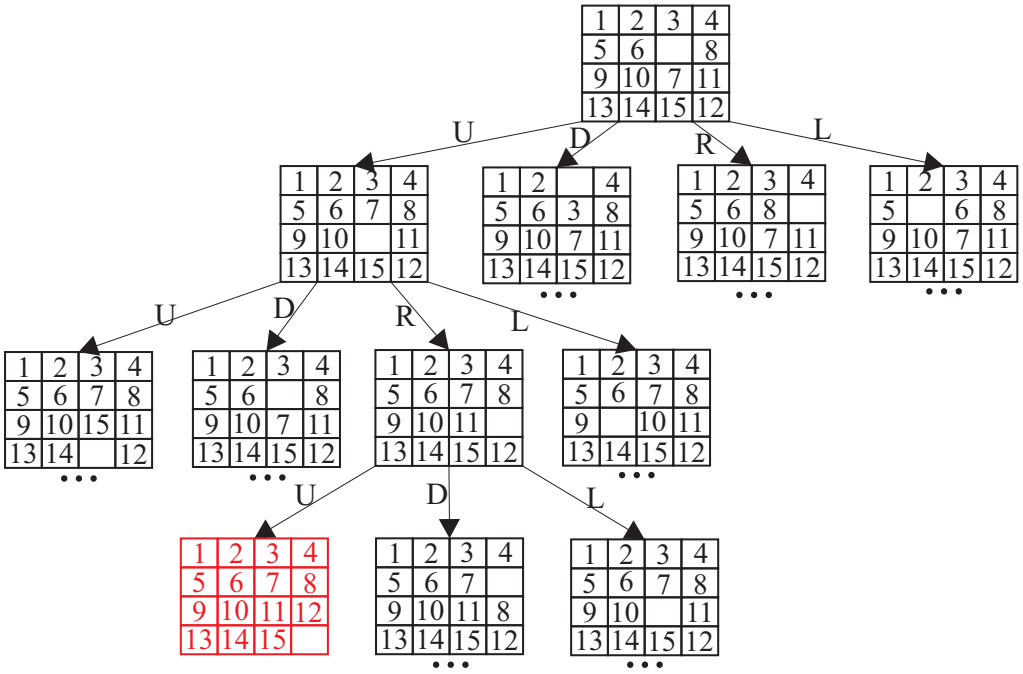


Figura 3.13: Obtenção da Configuração Final no Quebra-Cabeça do 15

Note também que cada movimento cria uma nova configuração ou estado. O

espaço de estados é formado por todas as configurações que podem ser alcançadas a partir de uma configuração inicial. Uma configuração inicial é um arranjo de 16 valores, no qual para a primeira posição da grade há 16 valores possíveis, para a segunda posição há 15 opções, para a terceira há 14 e assim segue até que na última posição há apenas 1 opção. Logo o número de configurações possíveis é 16!, das quais somente $\frac{16!}{2}$ são solucionáveis, já que uma configuração inicial só pode ser resolvida se existir um número definido de movimentos válidos que a leva à configuração alvo, ou ainda, se e somente se o valor $\sum_{i=1}^{16} menor(i)+r$ calculado da configuração inicial e alvo são de mesma paridade (ou seja, ou ambos são pares ou ambos impares). Onde $menor(i)$ é o número de quadrados $j$ tal que $j < i$ e posição$(j) >$ posição$(i)$ e $r$ é a coluna do espaço vazio (ROSSETTO, 2007).

Existem na literatura diversas soluções para o problema do Quebra-cabeça do 15 utilizando, por exemplo, algoritmos paralelos (GRAMA; KUMAR, 1999) e algumas variantes do algoritmo de busca $A^*$[3] (ROSSETTO, 2007). Sendo possível observar que a forma mais simples de resolver o problema é tratá-lo como um problema de busca clássico. A ideia básica é muito modesta: explorar todas as configurações que são acessíveis a partir da configuração inicial, e em algum lugar ao longo do caminho, espera-se encontrar a configuração alvo. Desta forma, se for mantido o controle de todos os movimentos realizados durante a exploração, a sequência de movimentos que deu origem à configuração alvo será conhecida. Se a configuração alvo não for encontrada, significa que não pode ser atingida a partir da configuração inicial em questão.

Nesta solução, o problema fundamental é ter certeza de que o algoritmo nunca ficará "ciclando", ou seja, preso em algum ciclo de configurações e não conseguir explorar todas as possíveis. Baseado nisto, o algoritmo 8 apresenta uma busca pelo espaço de estados. Caso a configuração alvo seja encontrada, a busca é encerrada

[3] O algoritmo $A^*$ é o algoritmo mais comum de busca heurística. Para cada nó gerado é calculado um valor que estima a sua distância em relação ao estado final. O estado que possuir o menor (ou maior) valor será expandido primeiro.

retornando sucesso, e se for encontrada uma configuração já visitada, ou seja, um limite de profundidade do grafo for atingido, a busca retrocede para o nó mais recente no caminho, que tenha irmãos ainda não visitados.

**Algoritmo 8:** quebracabeça15_backtracking(Configuração)

```
Entrada: Configuração
se Configuração = Configuração Alvo então
    retorne SUCESSO;
fim
T := Configuração;
enquanto Filhos[Configuração] <> 0 faça
    Filho := ProximoFilho;
    se Filho ∈ T então
        quebracabeça15_backtracking(Filho);
    fim
fim
retorne FALHA;
```

No algoritmo 8, $T$ é uma tabela de dispersão implementada de acordo com os preceitos discutidos no Capítulo 2, que armazena as configurações visitadas durante a busca. O uso da tabela de dispersão permite que o espaço alocado para armazenar as configurações visitadas seja linear, proporcional ao número de configurações visitadas. Além disso, depois de gerados os filhos, de acordo com os movimentos possíveis do quebra-cabeça, a verificação se o filho já havia sido visitado, pode ser realizada em tempo médio $O(1)$. Tal verificação é realizada constantemente pelo procedimento, pois após a geração dos filhos da configuração, de acordo com os movimentos possíveis, é feita esta verificação, retornando falha, em caso positivo, e fazendo o retrocesso para o nó anterior no caminho do grafo, a fim de identificar se há irmãos a serem visitados.

Quanto a complexidade de tempo, considere $B$ como sendo o número médio de filhos gerados por cada configuração e $m$, a profundidade máxima do grafo de espaço de estados. No pior caso, a complexidade do algoritmo 8 será $O(B^m)$.

# 4 EXPERIMENTOS

Na computação, existem diferentes métodos para identificar se um algoritmo está se comportando de forma eficiente. Um método facilmente executável, no qual podemos notar o desempenho do algoritmo em diversas situações, consiste na medição do tempo de execução computacional, que pode ser em segundos ou milissegundos. Este capítulo apresenta os resultados de desempenho dos algoritmos propostos para os problemas das seções [1][2] e [3] do Capítulo 3 em função do tempo de execução.

Todos os algoritmos foram implementados na linguagem C++, usando o compilador Bloodshed Dev C++ versão 4.9.9.2. Os testes foram realizados em um *notebook* com processador Pentium Dual-Core 2.3 GHz com 4.0 GB de RAM, com sistema operacional Windows 7 Home Basic 64 bits.

A análise de desempenho de algoritmos em função do tempo de execução é limitada parcialmente, devido a forte influência de variáveis como tempo e estabilidade do processamento, eficiência do compilador, entre outros. Para tentar minimizar os efeitos que esses limitantes causam nos resultados, foram coletadas cinco informações de tempo de cada um dos algoritmos e extraída a média aritmética. E apesar de na teoria não refletir com precisão o comportamento dos algoritmos, devido aos cuidados tomados, é possível notar que os tempos aferidos estão coerentes com as

complexidades apresentadas para cada problema no Capítulo 3.

As entradas utilizadas foram geradas por um procedimento de geração de números aleatórios disponível no Apêndice A. Todos os testes foram realizados com um fator de carga igual a 50%.

O problema dos pares $k$-complementares, foi o primeiro analisado. É importante observar que tanto a solução com força bruta, quanto o algoritmo *busca_ complementares()* que utiliza *hashing*, possuem complexidade igual a $O(n^2)$. Entretanto, como esperado devido a complexidade no caso médio, o algoritmo *busca_ complementares()* apresentou melhor desempenho conforme pode ser verificado na tabela 4.1, que mostra a média aritmética dos tempos de execução aferidos em milissegundos de ambos os algoritmos.

| **Tamanho da entrada** | **100** | **500** | **1000** | **5000** | **10000** | **50000** | **100000** |
|---|---|---|---|---|---|---|---|
| Força Bruta | 0 | 0 | 16 | 140 | 640 | 17862 | 44273 |
| *busca_ complementares()* | 0 | 0 | 15 | 47 | 243 | 5697 | 11245 |

Tabela 4.1: Tempo de execução do problema $k$-complementares em milissegundos

A tabela 4.2 mostra a média aritmética dos tempos de execução aferidos das soluções propostas para o problema da interseção de conjuntos em milissegundo. Foram considerados os algoritmos *interseção_ ordenada()* e *interseção_ hashing()*.

| **Tamanho da Entrada** | **100** | **500** | **1000** | **5000** | **10000** | **50000** | **100000** |
|---|---|---|---|---|---|---|---|
| *interseção_ ordenada()* | 0 | 0 | 16 | 137 | 393 | 3775 | 10779 |
| *interseção_ hashing()* | 0 | 0 | 15 | 125 | 371 | 3706 | 10687 |

Tabela 4.2: Tempo de execução do problema interseção de conjuntos em milissegundos

Observando a tabela 4.2 podemos notar que mesmo o algoritmo *interseção_ hashing()* possuindo complexidade no pior caso igual a $O(n^2)$, é possível executá-lo em um tempo menor que o algoritmo *interseção_ ordenada()* que possui comple-

xidade $O(n \log n)$. Isso porque a complexidade total esperada do algoritmo *interseção_hashing()* é $O(n)$.

Na tabela 4.3 é apresentada a média dos tempos de execução obtidos pelos testes realizados com os algoritmos de força bruta e *soma_de_funções_hashing()* que resolvem o problema da soma da igualdade de funções. Os tempos apresentados também estão expressos em milissegundos.

| **\|Entrada\|** | **10** | **40** | **80** | **100** | **200** | **300** | **400** | **500** |
|---|---|---|---|---|---|---|---|---|
| Força Bruta | 149 | 2855 | 8939 | 14820 | 41574 | 147357 | 387208 | 720477 |
| *soma...hashing()* | 63 | 1856 | 6022 | 10062 | 39003 | 91509 | 191677 | 250817 |

Tabela 4.3: Tempo de execução do problema da soma de funções em milissegundos

No problema da soma da igualdade de funções a complexidade das soluções está ligada ao número de quádruplas $(x, y, z, w)$ que satisfazem a condição de $f(x)+f(y) = f(z)+f(w)$, pois, por exemplo, nas situações em que a função em questão gera somente uma imagem para qualquer que seja a entrada, todas as imagens do conjunto irão atender a restrição e, portanto deverão ser exibidas. Logo, o número de quádruplas será igual a $n^4$ e o tempo total gasto pelo algoritmo que encontra as quádruplas não poderá ser menor que $O(n^4)$. Tal fato pode ser observado na tabela 4.3. Quando a entrada de tamanho 200 é computada, há uma grande quantidade de quádruplas que satisfazem a condição, então o tempo total gasto pelo algoritmo *soma_de_funções_hashing()* ficou próximo do tempo de execução do algoritmo de força bruta. Para as demais entradas, onde o número de quádruplas é menor, os tempos de execução observados para o algoritmo *soma_de_funções_hashing()* foi ligeiramente menor.

# 5 CONCLUSÃO

O emprego de estruturas de dados apropriadas é um dos pontos principais a se considerar na elaboração de algoritmos eficientes. Ocorre que determinadas estruturas, e não é o caso apenas das tabelas de dispersão, ficam de certa forma estigmatizadas, restritas a determinadas aplicações clássicas e suas variantes. É importante, contudo, saber aliar as ferramentas teóricas à criatividade na consideração do uso de estruturas eficientes, mesmo em aplicações que, à primeira vista, não pareciam sugeri-las.

Neste trabalho, ilustramos o uso de técnicas envolvendo *hashing* na concepção de soluções simples e eficientes para problemas pouco ortodoxos, ou em soluções pouco ortodoxas, mas eficientes, para problemas simples.

Também foram apresentados os desempenhos das soluções propostas para os problemas dos pares $k$-complementares, interseção de conjuntos e soma da igualdade de funções, de onde podemos concluir que as soluções com *hashing*, de fato apresentam bom desempenho na prática. Principalmente ao observarmos o desempenho das soluções do problema da interseção de conjuntos, no qual o algoritmo com *hashing* possui complexidade $O(n^2)$, contra $O(n \log n)$ do outro algoritmo proposto, e, apesar disso, o tempo de execução observado, ainda foi menor.

# REFERÊNCIAS

ASPNAS, M. ; SIGNELL, A. ; WESTERHOLM, J. Efficient assembly of sparse matrices using hashing. In: DANGARRA, J. ; MADSEN, K. ; WASNIEWISK, J. (Eds.). **Applied parallel computing. state of the art in scientific computing**. 8th International Workshop PARA 2006. Umea Sweden. Revised selected papers. Berlin Heidelberg, Springer-Verlag, 2007. p.900-907. (Lecture Notes in Computer Science, v. 4699).

BAI, Z. et al. **Templates for the solution of algebraic eigenvalue problems**: a practical guide. 1. ed. Philadelphia: SIAM Publication, 2000.

BALL, W. R. **Mathematical recreations and essays**. 10. ed. [London]: Limited Macmillan and CO., 1926.

CARTER, L. ; WEGMAN, M. N. Universal classes of hash functions. **Journal of Computer and System Sciences**, New York, v. 18, n. 2, p. 143-154, Apr. 1979.

CORMEN, T. H. et al. **Introduction to Algorithms**. 2th.ed. Cambridge, Mass.:The MIT Press, 2001.

CORWIN, E. ; LOGAR, A. Sorting in linear time - variations on the bucket sort. **Journal of Computing Sciences in Colleges**, North Newton, KS, v. 20, n. 1, p. 197-202, Oct. 2004.

DIETZFELBINGER, M. Design strategies for minimal perfect hash functions. **Lecture Notes in Computer Science,** v. 4665, n. 3, p. 2-17, 2007.

DROZDEK, A. **Data Structures and Algorithms in Java**. 1. ed. Pacific Grove: Brooks/Cole Pub., 2001.

GALLAGHER, R. H. Sparse matrices and their applications, Edited by Donald J. Rose and Ralph A. Willoughby, Plenum Press, New York, 1972. 215 p. **International Journal for Numerical Methods in Engineering**, Chichester, Eng., v. 4, n. 4, p. 600-600, Jul./Aug. 1972.

GAREY, M. R. ; JOHNSON, D. S. **Computers and intractability**: a guide to the theory of np-completeness. 1. ed. New York: W. H. Freeman & Co., 1979.

GEHAN, E. A. "Note on the 'Birthday Problem' ". **The American Statistician**. Ned Glick, v. 22, n. 2, p. 28, Apr. 1968.

GOODRICH, M. ; TAMASSIA, R. **Algorithm design. foundations,** analysis, and internet examples. Chichester.ed, Eng.: John Wiley & Sons Inc., 2001.

GRAMA, A. ; KUMAR, V. State of the art in parallel search techniques for discrete optimization problems. **IEEE Transactions on Knowledge and Data Engineering**, New York, v. 11, n. 1, p. 28-35, Jan. 1999.

HAYES, B. The easiest hard problem. **American Scientist**, New Haven, Conn., v. 90, n. 3, p. 113, May/Jun. 2002.

KNUTH, D. E. **The art of computer programming**: sorting and searching. 2. ed. Redwood City: Addison Wesley Longman Publishing Co., Inc., 1998. v. 3

MORETTIN, P. A. ; BUSSAB, W. O. **Estatística Básica**. 6. ed. São Paulo: Saraiva, 2010.

RATNER, D. ; WARMUTH, M. Finding a shortest solution for the $(n^2-1)$NxN extension of the 15-puzzle is intractable. In: AAAI 1986. NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 5., 1986, Philadelphia, PA .**Proceedings …** Philadelphia, PA: Morgan Kaufmann, 1986. v. 1. p. 168-172.

ROSSETTO, D. R. **Algoritmos de busca de caminhos em grafos aplicados aos problemas**: alinhamento de proteínas e quebra-cabeça de 15 peças. 2007. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Santa Catarina, Florianópolis, 2007.

SKIENA, S. S. ; REVILLA, M. A. **Programming challenges**. 1. ed. New York: Springer-Verlag, 2003.

SPYCHER, M. **High-throughput, thread-safe, LRU caching**. Out., 2011. Disponível em: <http://www.ebaytechblog.com/2011/08/30/high-throughput-thread-safe-lru-caching/>. Acesso em: 09 jan. 2012.

SZWARCFITER, J. L. **Grafos e algoritmos computacionais.** 1. ed. Rio de Janeiro: Campus, 1984.

SZWARCFITER, J. L. ; MARKENSON, L. **Estruturas de dados e seus algoritmos**. 2. ed. Rio de Janeiro: LTC, 1994.

TANENBAUM, A. S. **Sistemas operacionais modernos**. 2. ed. Upper Sadlle River, NJ: Prentice Hall, 2003.

TENENBAUM, A. A. ; LANGSAM, Y. ; AUGENSTEIN, M. J. **Estruturas de dados usando C**. 1. ed. São Paulo: Makron Books, 1995.

WIRTH, N. **Algorithms and data structures**. 2. ed. Englewood Cliffs: Prentice Hall, 1985.

ZIVIANI, N. **Projeto de algoritmos com implementações em pascal e C**. 4. ed. São Paulo: Pioneira, 1999.

# APÊNDICE A CÓDIGO FONTE DOS ALGORITMOS EM C++

Este apêndice apresenta o código fonte dos programas utilizados para realizar os testes cujos tempos de execução estão apresentados no Capítulo 4.

Programa usado para gerar os números aleatórios das entradas usadas nos testes:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main(){
  int ans,i;
  printf ("\nQuantos numeros devem ser gerados: ");
  scanf ("%d", &ans);
  ofstream escreve;
  escreve.open("numeros.txt");
  /* inicializar o gerador de números aleatórios com time(NULL)  */
  srand(time(NULL));
```

```
  for (i=0; i<ans; i++)
  {
        /* gera números aleatórios de 0 a 1000 */
        escreve<<rand() % 1000<<",";
  }
  escreve.close();
  system("PAUSE");
  return 0;
}
```

Programa para encontrar os pares $k$-complementares:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <string>
#include <vector>

/*Cria a estrutura */
typedef struct _elemento{
  int chave;
  struct _elemento *prox;
} elemento;

/*Cria a tabela hash */
#define HASHSIZE 10
static elemento* hashtable[HASHSIZE];
void inithashtable(){
```

```
  int i;
  for(i=0;i<HASHSIZE;i++)
    hashtable[i]=NULL;
}

/*Calcula o endereço-base*/
unsigned int hash(int s){
  unsigned int h=0;
  int i;
    h=s+h*19;
  return h%HASHSIZE;
}

/*Localiza chave */
elemento* lookup(int n){
  unsigned int hi=hash(n);
  elemento* np=hashtable[hi];
  for(;np!=NULL;np=np->prox){
    if(np->chave == n)
      return np;
  } return NULL;
}

/*Chama Localiza chave */
bool get(int chave){
  elemento* n=lookup(chave);
  return n!=NULL;
}
```

```
/*Insere chave na Tabela Hash */
int put(int chave){
  unsigned int hi;
  elemento* np;
  if(!get(chave)){
    hi=hash(chave);
    np=(elemento*)malloc(sizeof(elemento));
    if(np==NULL)
      return 0;
    np->chave=chave;
    if(np->chave==NULL) return 0;
    np->prox=hashtable[hi];
    hashtable[hi]=np;
  } return 1;
}

/*Imprime Tabela Hash*/
void mostratabela(){
  int i;
  elemento *t;
  for(i=0;i<HASHSIZE;i++){
    printf("%3d --> ",i);
if(hashtable[i]==NULL)
      printf("()");
    else{
      t=hashtable[i];
      for(;t!=NULL;t=t->prox) printf("(%d)",t->chave);
    } printf("\n");
  }
```

```
}

/*Imprime k-complementares usando Hashing*/
void kcomplementHash(int chaves[5], int k){
  int i,x,y;
  int j=0;
  elemento *t;
  for(i=0;i<5;i++){
    x=chaves[i];
    y=k-x;
    if(get(y)){
        printf("(%d,", x);
        printf("%d),", y);
    }
  }
}

/* Imprime k-complementares de conj[] e m é o número de elementos de conj[] */
int kcomplementVet(int conj[], int m, int k)
{
  int c=0; int i; int j;
  for(i=0;i<m;i++)
  {
    c=k-conj[i];
    for(j=0;j<m;j++)
    {
      if(c==conj[j])
      {
        printf("(%d,", conj[i]);
```

```
        printf("%d),", conj[j]);
      }
    }
  }
}

main(){
  int i; int conj[]={}; int k=0;
  int m = sizeof(conj)/sizeof(conj[0]);
  int chaves[]={}; int ans;
  printf ("\nDigite 1 para usar Vetor e 2 para usar Hashing: ");
  scanf ("%d", &ans);
  if(ans==1){
    clock_t startTime = clock();
    kcomplementVet(conj, m, k);
    clock_t endTime = clock();
    clock_t clockTicksTaken = endTime - startTime;
    clock_t timeInMSeconds = 1000*clockTicksTaken / (clock_t) CLOCKS_PER_SEC;
    printf("Tempo = %ld\n", timeInMSeconds);
  }
  if(ans==2){
    inithashtable();
    for(i=0;i<m;i++)
    { put(chaves[i]); }
    clock_t startTime = clock();
    kcomplementHash(chaves, k);
    clock_t endTime = clock();
    clock_t clockTicksTaken = endTime - startTime;
    clock_t timeInMSeconds = 1000*clockTicksTaken / (clock_t) CLOCKS_PER_SEC;
```

```
    printf("Tempo = %ld\n", timeInMSeconds);
  }
}
```

Funções que complementam o código fonte acima para encontrar a interseção de conjuntos:

```
/* Mergesort */
void merge(int vec[], int vecSize) {
  int mid; int i, j, k; int* tmp;
  tmp = (int*) malloc(vecSize * sizeof(int));
  if (tmp == NULL) { exit(1); }
  mid = vecSize / 2; i = 0; j = mid; k = 0;
  while (i < mid && j < vecSize) {
    if (vec[i] < vec[j]) {
      tmp[k] = vec[i]; ++i;
    }
    else {
      tmp[k] = vec[j]; ++j;
    } ++k;
  }
  if (i == mid) {
    while (j < vecSize) {
      tmp[k] = vec[j]; ++j; ++k;
    }
  }
  else {
    while (i < mid) {
      tmp[k] = vec[i]; ++i; ++k;
```

```
    }
  }
  for (i = 0; i < vecSize; ++i) {
    vec[i] = tmp[i];
  } free(tmp);
}

void mergeSort(int vec[], int vecSize) {
  int mid;
  if (vecSize > 1) {
    mid = vecSize / 2;
    mergeSort(vec, mid);
    mergeSort(vec + mid, vecSize - mid);
    merge(vec, vecSize);
  }
}

/*Imprime Intersecao de A e B usando Hashing */
void intersHash(int A[10]){
  int i,x,y; int j=0;
  elemento *t;
  for(i=0;i<10;i++){
    x=A[i];
    if(get(x)){
        printf("%d ", x);
    }
  }
}
```

```
/* Imprime interseção de conj1[] and conj2[] ordenado pelo Mergesort,
m é o número de elementos de conj1[],
n é o número de elementos de conj2[] */
int IntersVet(int conj1[], int conj2[], int m, int n)
{
  int i = 0, j = 0;
  mergeSort(conj1, m);
  mergeSort(conj2, n);
  while(i < m && j < n)
  {
    if(conj1[i] < conj2[j])
      i++;
    else if(conj2[j] < conj1[i])
      j++;
    else /* Se conj1[i] = conj2[j] */
    {
      printf(" %d ", conj2[j++]); i++;
    }
  }
}
```

Programa para encontrar a soma da igualdade de funções

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <math.h>
```

```
/*Cria a estrutura */
typedef struct _elemento{
  int chave; int val1; int val2; struct _elemento *prox;
} elemento;

/*Cria a tabela hash */
#define HASHSIZE 20
static elemento* hashtable[HASHSIZE];
void inithashtable(){
  int i;
  for(i=0;i<HASHSIZE;i++){
    hashtable[i]=NULL;
  }
}

/*Calcula o endereço-base*/
unsigned int hash(int s){
  unsigned int h=0;
  int i;
    h=s+h*19;
  return h%HASHSIZE;
}

/*Insere chave na Tabela Hash */
int put(int chave, int val1, int val2){
  unsigned int hi; elemento* np; hi=hash(chave);
  np=(elemento*)malloc(sizeof(elemento));
  if(np==NULL)
    return 0;
```

```
  np->chave=chave;
  /*Salva as entradas*/
  np->val1=val1; np->val2=val2;
  if(np->chave==NULL) return 0;
  np->prox=hashtable[hi];
  hashtable[hi]=np;
  return 1;
}

/*Imprime Tabela Hash*/
void mostratabela(){
  int i; elemento *t;
  for(i=0;i<HASHSIZE;i++){
    printf("%3d --> ",i);
if(hashtable[i]==NULL)
      printf("()");
    else{
      t=hashtable[i];
      for(;t!=NULL;t=t->prox){
        printf("(%d{",t->chave); printf("%d,",t->val1); printf("%d})",t->val2);
      }
    } printf("\n");
  }
}

/*Imprime x,y,z,w tal que f(x)+f(y)=f(z)+f(w) usando hashing*/
void somafuncaoHash(){
  int i,x,y; elemento *t; elemento *t2;
  for(i=0;i<HASHSIZE;i++){
```

```
    if(hashtable[i]!=NULL){
      t=hashtable[i]; t2=hashtable[i];
      while(t!=NULL){
        if(t->chave == t2->chave){
          printf("(%d,",t->val1); printf("%d,",t->val2);
          printf("%d,",t2->val1); printf("%d)",t2->val2);
        } t2=t2->prox;
        if(t2==NULL){
          t=t->prox; t2=t;
        }
      }
    }
  }
}

/* Imprime x,y,z,w tal que f(x)+f(y)=f(z)+f(w) e
m é o número de entradas de uma função f */
int somafuncaoVet(int entradas[10], int imagens[10], int col)
{
  int i, j, k, l;
  for(i=0;i<col;i++){
    for(j=0;j<col;j++){
      for(k=0;k<col;k++){
        for(l=0;l<col;l++){
          if(imagens[i]+imagens[j]==imagens[k]+imagens[l]){
            printf("(%d,", entradas[i]); printf("%d,", entradas[j]);
            printf("%d,", entradas[k]);  printf("%d),", entradas[l]);
          }
        }
```

```
      }
    }
  }
}

/*Principal */
main(){
  int i; int j; int soma; int entradas[10] ={};
  int m = sizeof(entradas)/sizeof(entradas[0]);
  int imagens[m];
  for(i=0;i<m;i++)
  {
     imagens[i]=((entradas[i]*entradas[i])-(entradas[i]))+19;
  }
  int ans;
  printf ("\nDigite 1 para usar Vetor e 2 para usar Hashing: ");
  scanf ("%d", &ans);
  if(ans==1){
    clock_t startTime = clock();
    somafuncaoVet(entradas,imagens,m);
    clock_t endTime = clock();
    clock_t clockTicksTaken = endTime - startTime;
    clock_t timeInMSeconds = 1000*clockTicksTaken / (clock_t) CLOCKS_PER_SEC;
    printf("Tempo = %ld\n", timeInMSeconds);
  }
  if(ans==2){
     clock_t startTime = clock();
     inithashtable();
     for(i=0;i<m;i++){
```

```
      for(j=0;j<m;j++){
        soma= imagens[i]+imagens[j];
        put(soma,entradas[i],entradas[j]);
      }
    }
    somafuncaoHash();
    clock_t endTime = clock();
    clock_t clockTicksTaken = endTime - startTime;
    clock_t timeInMSeconds = 1000*clockTicksTaken / (clock_t) CLOCKS_PER_SEC;
    printf("Tempo = %ld\n", timeInMSeconds);
  }
  system("PAUSE");
  return 0;
}
```