

Estrutura de Dados I

Capítulo 11: Dicionários e Tabelas de Símbolos

Introdução

- **Dicionários** são TADs que permitem o armazenamento e a busca de dados pelo próprio **conteúdo dos dados**, através de pares de mapeamento:

chave -> valor

(podemos dizer que um dicionário é um TAD para chave-valor)

- Os dicionários diferenciam-se dos containers pois estes não consideram o conteúdo dos dados para suas operações (pilha, fila), e aqueles consideram o conteúdo (chaves e valores)
- Os principais dicionários na computação são:
 - **Dictionary** (map, associative array)
 - **Symbol Table**

Introdução

- **Dictionary** (map, associative array)

- **TAD genérico** para armazenar e recuperar pares “**chave:valor**”
- **Chave** e **Valor** são **genéricos**
- **Semântica**: armazena valores genéricos
- **Uso amplo**: caches, contagens de frequência, roteamentos, configurações, etc.
- **Mais abstrato**: não impõe nada sobre o significado de chaves e valores, apenas que sejam comparáveis

- **Symbol table**

- **TAD especializado** para armazenar e recuperar pares “**chave:valor**” (é, na verdade, um dictionary) com uso especializado em compiladores e interpretadores
- **Chave**: normalmente é um **identificador**:
 - Nome de variável, nome de função, nome de constante, etc.
- **Valor**: normalmente são os **atributos** associados ao identificador:
 - Tipo, escopo, endereço de memória, tamanho, valor constante, qualificadores, etc.
- **Semântica**: armazena metadados de símbolos de um programa
- **Escopo e níveis**: costuma lidar com múltiplos níveis de escopo (variáveis locais e globais)
- **Uso em compiladores e interpretadores**: análise léxica, análise sintática, análise semântica, validação de tipos, geração de código, resolução de nomes em múltiplos módulos

Introdução

- **Comportamentos comuns:**

- `insert(dictionary, key, info)`
- `lookup(dictionary, key)`
- `remove(dictionary, key)`

- **Outros comportamentos (menos comuns):**

- `keys(dictionary)`
- `values(dictionary)`
- `iterate(dictionary)`

Introdução

- **Estruturas de dados comuns para a implementação:**
 - **Árvores balanceadas:** lookup, insert e remove em $O(\log n)$
 - AVL
 - Red-Black
 - **Árvore Trie** (árvore de prefixos): operações em $O(\ell)$, onde ℓ = tamanho da chave
 - **Hash table:** operações em $O(1)$ na média, e em $O(n)$ no pior caso
 - É a implementação mais freqüente em compiladores modernos
- **Estruturas de dados menos comuns para a implementação:**
 - **Arrays:** inserir é $O(n)$ e lookup $O(\log n)$ se usar busca binária

11.1 Definição do TAD Dicionário: `dicionarioTAD.h`

- Objetivo: definir uma abstração para um dicionário que seja flexível o suficiente para ser usado em uma ampla variedade de aplicações, mantendo um alto nível de eficiência na implementação.
- Na interface teremos, como sempre, o tipo abstrato declarado como um ponteiro para um tipo concreto que será declarado e definido apenas na implementação:

```
/**  
 * Tipo: dicionarioTAD  
 * -----  
 * Este é o TAD usado para representar um dicionário.  
 */
```

```
typedef struct dicionarioTCD *dicionarioTAD;
```

11.1 Definição do TAD Dicionário: dicionarioTAD.h

- Precisamos declarar os comportamentos do dicionário:

```
/**  
 * Função: criar_dicionario  
 * Uso: dic = criar_dicionario( );  
 * -----  
 * Esta função aloca um novo dicionário na memória, sem nenhuma entrada definida  
 * e retorna o ponteiro para o dicionário. Se o dicionário não puder ser criado,  
 * retorna NULL.  
 */  
  
dicionarioTAD criar_dicionario (void);
```

11.1 Definição do TAD Dicionário: dicionarioTAD.h

- Precisamos declarar os comportamentos do dicionário:

```
/**
 * Procedimento: remover_dicionario
 * Uso: remover_dicionario(dic);
 * -----
 * Este procedimento libera todas as áreas de armazenamento em memória alocadas
 * para o dicionário. Note que o procedimento recebe um PONTEIRO para um
 * dicionário (dicionarioTAD), ou seja, recebe um ponteiro para ponteiro para
 * struct dicionarioTCD.
 */

void remover_dicionario(dicionarioTAD *dic);
```


11.1 Definição do TAD Dicionário: `dicionarioTAD.h`

- Em relação aos dois comportamentos mais importantes de um dicionário, temos um problema:

```
inserir(dicionario, chave, valor);
```

```
valor = procurar(dicionario, chave);
```

- Qual o tipo de dados para a chave?**
- Qual o tipo de dados para o valor?**

Queremos 2 coisas:

- a) Maior flexibilidade possível para o cliente; e
- b) Maior eficiência possível na implementação

11.1 Definição do TAD Dicionário: `dicionarioTAD.h`

- A representação do **valor** é mais fácil. Considere o seguinte:
 - O valor é um **conceito totalmente controlado pelo cliente**
 - Seu dicionário deve permitir que o cliente armazene qualquer coisa que ele queira
 - De sua perspectiva como o implementador do dicionário, **você não liga para o tipo de dado pois você nunca fará nada com o valor**, apenas fará o armazenamento e depois retornará esse valor quando o cliente buscar pela chave
 - Assim, o tipo usado para representar os valores deve ser **o mais geral possível**.
- Ainda temos que considerar o seguinte:
 - Dicionários são muito mais freqüentes do que pilhas e filas. É comum que um mesmo cliente utilize diversos dicionários ao mesmo tempo, cada um para armazenar valores diferentes.
 - O dicionário deve ter um **valor sentinela** para indicar que uma busca não encontrou nenhum resultado (a chave não existe no dicionário). O sentinela deve ter o mesmo tipo do valor e se usarmos tipos como, por exemplo, `int` para os valores não teremos nenhum exclusivo que pode ser usado como sentinela.

11.1 Definição do TAD Dicionário: `dicionarioTAD.h`

- Portanto, o melhor tipo de dado para o **valor** é o tipo **`void *`**
- Usar o tipo ponteiro para void permite que o cliente armazena qualquer valor, usando ponteiros apropriados.
- Usar o tipo ponteiro para void permite que a função de busca retorne um sentinela adequado para indicar que uma determinada chave não está definida no dicionário (não existe no dicionário).
- Usar o tipo ponteiro para void cria um pequeno overhead no cliente, mas dá muito mais flexibilidade.

11.1 Definição do TAD Dicionário: `dicionarioTAD.h`

- A representação do **chave** é mais difícil. Considere o seguinte:
 - A implementação trata de modo diferente a chave e o valor:
 - O valor é apenas armazenado e devolvido e, portanto, a implementação não precisa saber nada sobre esse valor e não precisa fazer nada com esse valor
 - A chave precisa ser conhecida e manipulada pela implementação. No mínimo a implementação precisa verificar se a chave existe e, se existir, a que valor ela corresponde. Para fazer isso a implementação precisa, no mínimo, ser capaz de comparar uma chave com outra. Isso **exclui**, em princípio, o tipo `void *`.
- A abordagem mais utilizada para as chaves é usar o tipo **string** (`char *`):
 - Permitem muita flexibilidade
 - É fácil converter outros tipos em strings

11.1 Definição do TAD Dicionário: dicionarioTAD.h

- Comportamentos de inserção e busca:

```
/**
 * Procedimento: inserir
 * Uso: inserir(dicionario, chave, valor);
 * -----
 * Este procedimento recebe um dicionário (dicionarioTAD), uma chave (string) e
 * um valor (void *), e associa a chave com o valor no dicionário. Cada chamada
 * de "inserir" substitui qualquer definição anterior para a chave. Nesta
 * interface a chave não pode ser nula (NULL): se o usuário passar uma chave
 * NULL a operação de inserção não tem efeito nenhum.
 */

void
inserir (dicionarioTAD dic, string chave, void *valor);
```

11.1 Definição do TAD Dicionário: dicionarioTAD.h

- Comportamentos de inserção e busca:

```
/**
 * Função: procurar
 * Uso: valor = procurar(dicionario, chave);
 * -----
 * Esta função recebe um dicionário (dicionarioTAD) e uma chave (string), e
 * retorna o valor associado à chave no dicionário. Se a chave não estiver
 * definida no dicionário, retorna NULL.
 */

void *
procurar (dicionarioTAD dic, string chave);
```

11.1 Definição do TAD Dicionário: `dicionarioTAD.h`

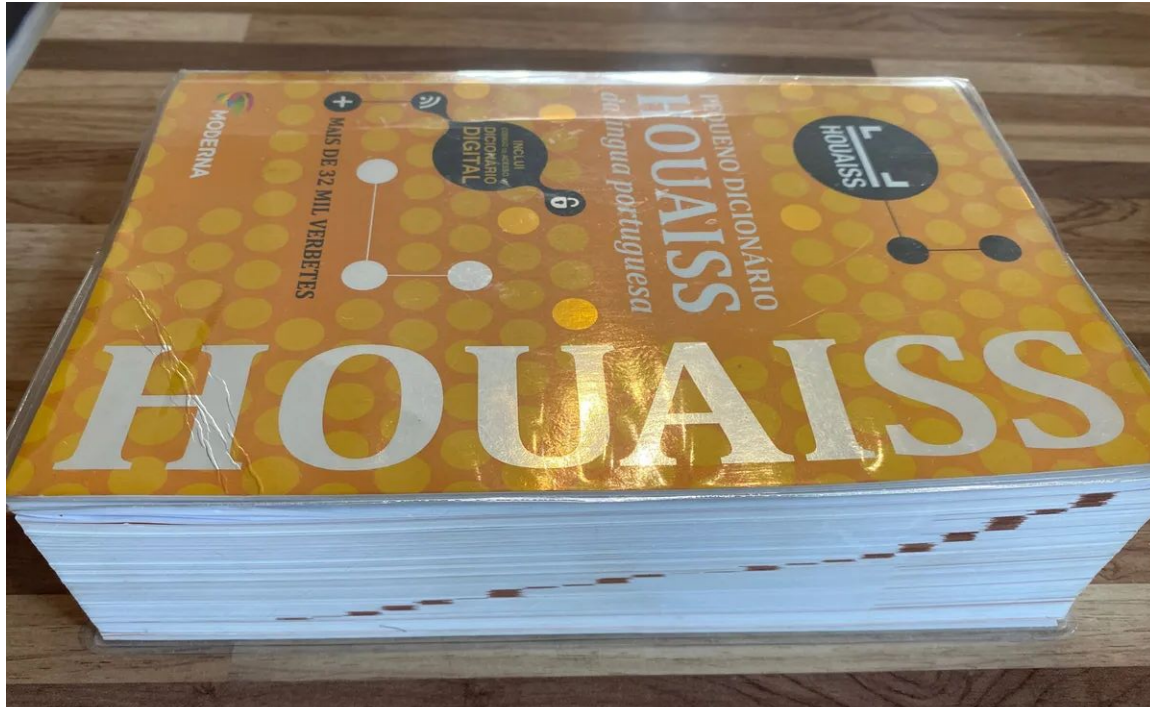
- Importante:
 - Se sua implementação **não permitir que o usuário use um NULL value como chave**, então você pode usar um NULL value como sentinela para indicar que alguma chave buscada não está definida no dicionário (a chave buscada não existe). Esse é o comportamento mais comum em C.
 - Se sua implementação **permitir que o usuário use um NULL value como chave**, então você terá que definir um valor especial, como o **UNDEFINED** de `genlib.c`, como sentinela para indicar que alguma chave buscada não está definida no dicionário. Isso é possível em Java e Python. Em C++ há algumas bibliotecas que permitem.

11.2 Hashing e hash table

- Dicionários são tão usados na computação que a **eficiência da implementação é crítica**:
 - Mesmo implementações $O(\log n)$ podem não ser suficientes.
 - O que queremos é uma implementação de tempo constante, pelo menos na média dos casos, para as operações de inserção e de busca. Para isso usamos **hashing**.
- **Hashing**: é uma estratégia que consiste em **mapear chaves** (strings, por exemplo) **para números inteiros** e, então, usar esses números inteiros como índices para localizar as chaves de modo rápido em um array.
 - O hash é amplamente utilizado em aplicações e merece reconhecimento como uma das invenções mais inteligentes da ciência da computação.

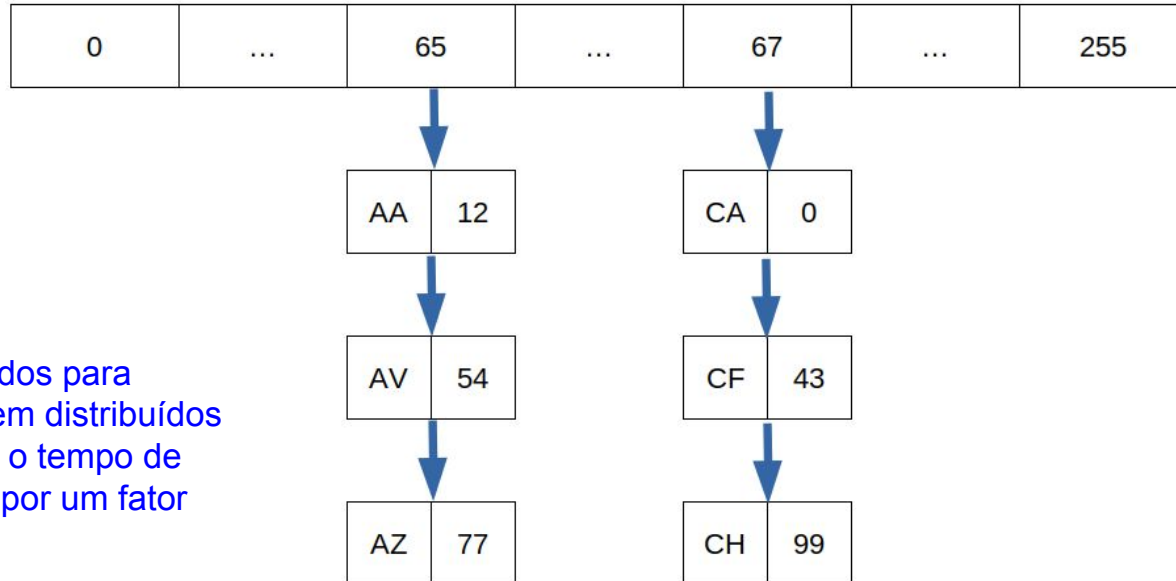
11.2 Hashing e hash table

- Por que o hashing é necessário?
 - Estratégia do dicionário físico:



11.2 Hashing e hash table

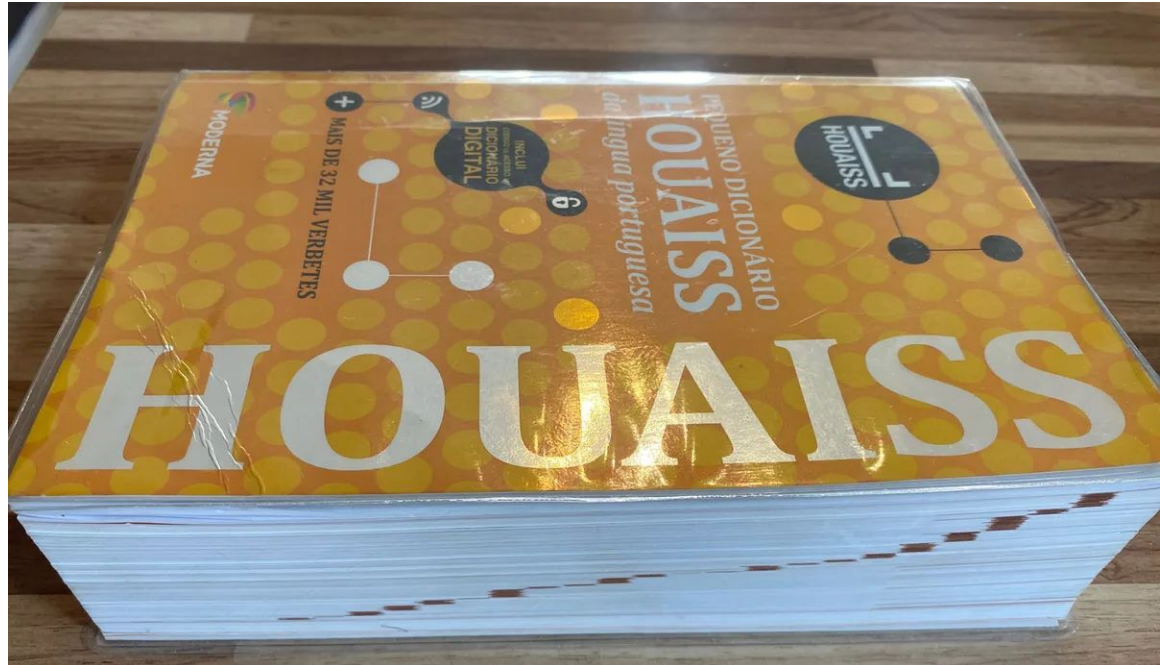
- Por que o hashing é necessário?
 - Mimetizando a estratégia do dicionário físico:
 - Vamos dividir o dicionário em 256 listas independentes de pares chaves/valor, uma lista para cada caractere do início da chave.



Se os caracteres usados para formar as chaves forem distribuídos de maneira uniforme, o tempo de busca seria reduzido por um fator de 256!

11.2 Hashing e hash table

- Por que o hashing é necessário?
 - A estratégia do dicionário físico é ruim!
 - Algumas letras têm mais chaves do que outras.
 - Algumas listas ficariam vazias e outras teriam muitas e muitas chaves
 - A eficiência dependeria de quão comum é a primeira letra da chave.



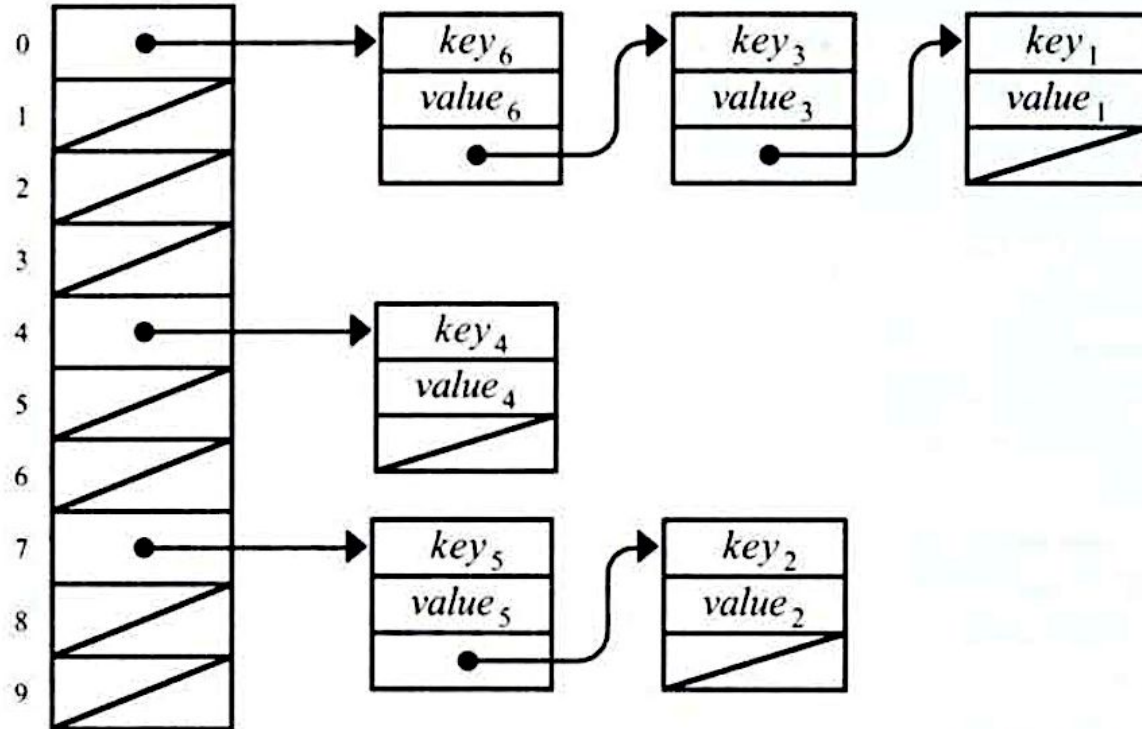
11.2 Hashing e hash table

- O que nós precisamos é de alguma função F que transforme uma chave em um índice inteiro uniformemente distribuído que selecione uma das listas internas. Por exemplo: se $F(\text{chave}) = 17$, as funções de inserção e busca tentarão inserir ou buscar essa chave apenas na lista 17.
- A função F , que reduz a chave à um inteiro em uma faixa fixa uniformemente distribuída é chamada de **função hash**.
- O valor da função hash para uma chave particular é o **código hash**.
- A **estrutura de dados** resultante é chamada de **hash table**.

11.2 Hashing e hash table

- Tabela Hash:

- Array de listas encadeadas, cada elemento do array é chamado de **bucket**.
- A função hash recebe uma chave e obtém o **hash code**, o índice do array para um dos buckets.
- Cada bucket contém um **ponteiro para a 1ª célula** da lista de pares chaves/valor.
- Todas as células em uma lista têm o mesmo hash code (**colisão**: duas ou mais chaves como mesmo hash).




11.2 Hashing e hash table

- A **função hash é crítica nesse processo**, pois ela deve receber as mais diferentes chaves possíveis, e deve mapear para um **hash code que é uniformemente distribuído**, ou seja, a função hash deve distribuir os pares chaves/valor entre os buckets de forma que o tamanho da lista encadeada apontada por cada bucket seja aproximadamente o mesmo, **minimizando as colisões**.
- **A escolha da função hash NÃO É FÁCIL**, existem inúmeras maneiras de implementar. Existem linhas de pesquisa avançadas para a escolha de uma boa função hash.
- Usaremos um método chamado de **congruência linear** para implementar uma função de hash.

11.2 Hashing e hash table

Diminui a probabilidade de colisão entre chaves semelhantes para, praticamente, ao acaso. Discussão muito AVANÇADA para nós no momento.



```
#define MULTIPLICADOR ((unsigned long int) -1664117991L)

/**
 * Função: hash
 * Uso: hash_code = hash(chave, buckets);
 * -----
 * Esta função recebe uma chave e retorna um hash code que será um número
 * inteiro (size_t) entre 0 e (buckets - 1). O parâmetro "buckets" corresponde
 * ao número total de buckets da tabela hash. O cálculo do hash code é feito
 * utilizando-se um método chamado de congruência linear. A escolha do valor do
 * MULTIPLICADOR pode ter um efeito significativo na performance do algoritmo,
 * mas não em sua corretude.
 */

size_t hash (const string chave, size_t buckets)
{
    unsigned long int hashcode = 0;

    for (size_t i = 0; chave[i] != '\0'; i++)
        hashcode = hashcode * MULTIPLICADOR + (unsigned char) chave[i];

    return (hashcode % buckets);
}
```


11.2 Hashing e hash table

- Outra coisa que precisamos levar em consideração é o **número de buckets**, pois a probabilidade de colisão também dependerá disso.
- As **colisões diminuem a eficiência da tabela hash** pois as buscas e inserções precisarão ser feitas em longas listas encadeadas.
- Como nosso objetivo é alcançar operações $O(1)$ na média dos casos, é importante que **as listas encadeadas permaneçam razoavelmente pequenas** e, para isso, precisamos garantir que a quantidade de buckets seja grande o suficiente para diminuir as colisões e manter as listas pequenas.

11.2 Hashing e hash table

- Se a função hash realmente está fazendo um bom trabalho e distribuindo uniformemente as chaves entre os buckets, então o tamanho médio de cada lista encadeada é dado por:

$$\lambda = \frac{N_{\text{entries}}}{N_{\text{buckets}}}$$

Se a quantidade total de entradas for 3 vezes maior do que o número de buckets, então cada lista terá 3 pares, o que significa que 3 operações de comparação de strings serão necessárias, em média, para encontrar uma chave.

- Esse número é o **fator de carga** da hash table, e deve ser mantido baixo (basta aumentar o número de buckets, mesmo que alguns fiquem vazios).

11.2 Hashing e hash table

- Determinar a quantidade correta de buckets É DIFÍCIL, e nenhum número funcionará bem em todas as situações.
- Mesmo que o número de buckets seja grande, se o cliente colocar mais e mais pares chave/valor, a performance irá diminuir. Se quisermos restaurar a performance nesse caso, podemos aumentar dinamicamente o número de buckets. Isso é chamado de **rehashing**. O problema é que ao aumentar o número de buckets, todos os hash code serão alterados e teremos que recalcular e realocar todas as células. Esse processo pode demorar, mas é feito raramente e, por isso, não causa muito impacto na aplicação.

11.2 Hashing e hash table

- Determinar a quantidade correta de buckets É DIFÍCIL, e nenhum número funcionará bem em todas as situações.
- Mesmo que o número de buckets seja grande, se o cliente colocar mais e mais pares chave/valor, a performance irá diminuir. Se quisermos restaurar a performance nesse caso, podemos aumentar dinamicamente o número de buckets. Isso é chamado de **rehashing**. O problema é que ao aumentar o número de buckets, todos os hash code serão alterados e teremos que recalcular e realocar todas as células. Esse processo pode demorar, mas é feito raramente e, por isso, não causa muito impacto na aplicação.

11.2 Hashing e hash table

- Uma última observação: a maioria das funções hash funcionam melhor se **o número de buckets for PRIMO**.
- Se o número de buckets não for primo, padrões que causam desbalanceamento da distribuição das chaves têm maior probabilidade de ocorrer, aumentando as colisões.
- Assim, em resumo:
 - Use boas funções de hash
 - Use uma quantidade suficiente de buckets para manter o fator de carga baixo
 - Use um número PRIMO de buckets.