Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • IMD0030
◁ Exercícios de Implementação de Hash Table ▷
2 de maio de 2016

Objetivos

O objetivo deste exercício é oferecer uma oportunidade para implementar a estrutura de dados *tabela de dispersão* em C++. Informações sobre o funcionamento de uma tabela de dispersão, funções de dispersão e estratégias para tratamento de colisões devem ser procurados em livros de estrutura de dados como [2] ou [1].

Sumário

1 A Aplicação Exemplo: Cadastro de Contas Corrente

Para motivar o desenvolvimento da tabela de dispersão vamos considerar que desejamos criar uma aplicação simples que mantém um cadastro de *contas corrente*. Os dados das contas corrente serão armazenados em uma **tabela de dispersão** de tamanho a ser definido pelo cliente[1]. A aplicação deve suportar operações básicas como *inserir*, *consultar* e *remover* constas correntes. No caso das operações de consulta e remoção, precisamos definir de que maneira vamos identificar cada conta corrente, ou seja, qual a **chave** utilizada. Mas isso será explicado mais adiante.

[1] Na verdade, dentro da classe devemos encontrar o primeiro número primo maior que o tamanho fornecido pelo cliente; este vai ser o tamanho real da tabela de dispersão.

Inicialmente devemos modelar o comportamento de uma conta corrente através de uma `struct` do C++. Lembre-se que uma `struct` possui o mesmo status que uma `class`. A diferença entre as duas é que por *default* os métodos e membros de um `struct` são públicos, enquanto que os de uma `class` são privados.

O tipo conta corrente, denominado de `Account`, possui os seguintes *membros*: nome do cliente, código do banco, número da agência, número da conta e saldo. Para ilustrar o conceito de uma conta corrente, considere os exemplos da Tabela 1.

Id	Nome cliente	Banco	Agência	Conta	Saldo
1	*"Alex Bastos"*	1	1668	54321	R$ 1500.00
2	*"Aline Souza"*	1	1668	45794	R$ 530.00
3	*"Cristiano Ronaldo"*	13	557	87629	R$ 150000.00
4	*"Jose Lima"*	18	331	1231	R$ 850.00
5	*"Saulo Cunha"*	116	666	84312	R$ 5490.00

Tabela 1: Exemplo de contas corrente.

Além dos *membros* (públicos) definidos anteriormente, a classe possui um construtor simples que apenas inicializa seus membros e um método denominado de `getKey()` que retorna a **chave** associada a uma conta corrente.

Considerações sobre a Chave

Note que a definição de uma chave depende exclusivamente da aplicação e de sua finalidade. O importante é que *uma chave deve identificar unicamente cada elemento em uma coleção*; em outras palavras, não podemos ter chaves repetidas. Neste exercício, trabalharemos com 3 versões de chaves:

Versão 1: Uma chave composta por um inteiro correspondente ao número da conta corrente.

```
using AcctKey = int;
```

Versão 2: Uma chave composta de dois campos definidos por um `std::pair`, que contém o nome do cliente e o número da conta.

```
using AcctKey = std::pair< std::string, int >;
```

Versão 3: Uma chave composta por meio de 4 campos definidos por uma `std::tuple`, que contém o nome do cliente, o código do banco, o número da agência e o número da conta.

```
using AcctKey = std::tuple< std::string, int, int, int >;
```

Estas versões de chave têm o propósito apenas didático. Começamos com uma chave simples na **Versão 1**. Na vida real dificilmente esta chave seria válida, visto que provavelmente não seria capaz de identificar unicamente uma conta corrente; é perfeitamente possível que contas corrente em bancos distintos possuam um mesmo número! A chave da **Versão 2** é um pouco melhor, pois associa

o nome do cliente ao número da conta; porém ainda é possível existir contas corrente diferentes que tenham o mesmo nome de cliente e mesmo número (em bancos diferentes). A chave da **Versão 3** é a melhor das três, visto que caracteriza unicamente uma conta corrente por meio do nome do cliente, banco, agência e número de conta corrente; não é possível existir contas corrente distintas com uma mesma chave proposta na **Versão 3**[2].

2 A TAD Tabela de Dispersão

A Tabela de Dispersão usada no exercícios é denominada de `HashTbl`. A `HashTbl` será baseada na aplicação de *dispersão dupla*. Uma dispersão dupla significa que aplicaremos duas funções de dispersão, uma função (substituível) definida pelo cliente e outra função (fixa) definida pela própria `HashTbl`. A primeira depende da aplicação e deverá ser passada para a `HashTbl` por meio de argumento-template, explicado mais adiante na Seção 3.3. A segunda função de dispersão será baseada no *método da divisão*, ou seja, a função calcula o resto da divisão inteira entre um número, produzido pela função de dispersão primária, e o tamanho da tabela; o resultado é um inteiro na faixa $[0; tam - 1]$, onde tam seria o tamanho máximo da tabela.

Em resumo, temos a Equação

$$\text{end}_{\text{tabela}} = f_s(f_p(\texttt{AcctKey})) \tag{1}$$

onde $\text{end}_{\text{tabela}}$ é o endereço da tabela produzido pela dispersão, ou seja, um índice em $[0; tam-1]$; f_s é a função de dispersão secundária (fixa); f_p é a função de dispersão primária, definida pelo cliente e que recebe uma `AcctKey` e produz um número inteiro sem sinal, `std::size_t`, para manter compatibilidade com outras função de dispersão do STL.

A seguir serão apresentados dois exemplos para ilustrar o cálculo de um endereço em uma tabela de dispersão de tamanho 23, para as duas primeiras versões de chaves definidas na Seção 1.

2.1 Dispersão Dupla sobre a Chave da Versão 1

Suponha que a aplicação cliente decide que a função de dispersão secundária para a **Versão 1** será a função de dispersão padrão do STL que recebe um inteiro: `std::hash<int>`. A função `std::hash<int>` mapeia um número inteiro para um *inteiro sem sinal*. Veja abaixo o exemplo de uma possível saída para esta função hash:

```
1 #include <functional>
2 ...
3 std::cout << "Hash (123) = " << std::hash<int>()( 123 ) << std::endl;
4 std::cout << "Hash (-123) = " << std::hash<int>()( -123 ) << std::endl;
```

que resulta em

```
Hash (123) = 123
Hash (-123) = 18446744073709551493
```

[2]Uma aplicação real provavelmente incorporaria o CPF do cliente à composição da chave, pois é um identificador único natural.

Ao aplicarmos a função de dispersão dupla definida na Equação 1 para uma tabela de dispersão com capacidade para 23 entradas, obteremos o resultado apresentado na Tabela 2.

Mapeamento por Dispersão Dupla

Id Conta	**Chave (nº conta)**	$f_p =$ `std::hash<int>(Chave)`	**Endereço final:** $f_s = f_p\%23$
1	54321	54321	18
2	45794	45794	1
3	87629	87629	22
4	1231	1231	12
5	14312	84312	15

Tabela 2: Aplicação da dispersão dupla na chave da **Versão 1** para uma tabela de tamanho 23. Note que a função de dispersão primária recebe um inteiro e retorna um inteiro *sem sinal*.

2.2 Dispersão Dupla sobre a Chave da Versão 2

Lembre que a chave definida na **Versão 2** é formada pelo par *nome do cliente* e *número da conta*, ou seja, espera-se que este par de informações seja suficiente para identificar unicamente uma conta corrente.

Suponha agora que a aplicação cliente decide que a função de dispersão secundária para a **Versão 2** será realizada da seguinte forma: (1) aplicar a função de dispersão padrão do STL que recebe um `std::string` sobre o *nome do cliente da conta corrente*, (2) aplicar a função de dispersão padrão do STL que recebe um inteiro sobre o *número da conta*, e (3) combinar os resultados anteriores com a operação lógica *ou exclusivo* sobre seus bits.

```
f_p = std::hash<std::string>()( Nome_cliente ) xor std::hash<int>()( N°_Conta ) ;
```

O resultado é apresentado na Tabela 3.

Mapeamento por Dispersão Dupla

Id Conta	**Chave (nome + nº conta)**	f_p	**Endereço final**
1	(*"Alex Bastos"* +54321)	7509724456891295127 `xor` 54321	$f_p\%23 = 1$
2	(*"Aline Souza"* +45794)	8135930721642426 `xor` 45794	$f_p\%23 = 2$
1	(*"Cristiano Ronaldo"* +87629)	6805834961756837907 `xor` 54321	$f_p\%23 = 13$
4	(*"Jose Lima"* +1231)	17264114507666041099 `xor` 1231	$f_p\%23 = 13$
5	(*"Saulo Cunha"* +84312)	7466768777966487320 `xor` 14312	$f_p\%23 = 16$

Tabela 3: Aplicação da dispersão dupla na chave da **Versão 2** para uma tabela de tamanho 23. O símbolo `xor` representa a operação *ou exclusivo* bit-a-bit.

3 Implementando `HashTbl`

A tabela de dispersão representada pela classe `HashTbl` é constituída por um vetor alocado dinamicamente de ponteiros para listas encadeadas de colisão. Um endereço primário da tabela, portanto, é um apontador para uma lista de *itens de tabela*, `HashEntry`. Um item de tabela, por sua vez, é um par *chave* e *informação*, ou seja, um **mapeamento** entre uma chave (única) e a informação que desejamos armazenar.

Note que tanto a *informação* quanto a *chave* são tipos de dados definidos pelo cliente, ou seja, são dependentes da aplicação que se deseja desenvolver. No nosso estudo de caso, temos três versões de chave (ver Seção 1) e a informação que desejamos armazenar na tabela é um objeto do tipo *conta corrente*, ou `Account`. Sendo assim, podemos definir um item de tabela como:

```
1 template< class KeyType, class DataType >
2 class HashEntry {
3     public:
4         HashEntry ( KeyType _k, DataType _d ) : mKey( _k ), mData( _d )
5         { /* Empty */ };
6         KeyType mKey;   /*!< Stores the key for an entry. */
7         DataType mData; /*!< Stores the data for an entry. */
8 };
```

Esta classe será utilizada principalmente pela classe `HashTbl`, ou seja, o cliente não precisa tomar conhecimento dela. Toda a comunicação entre o cliente e a tabela de dispersão será realizada pela interface pública da tabela de dispersão, apresentada a seguir.

3.1 Interface de `HashTbl`

```
1  template < typename KeyType, typename DataType >
2  class HashTbl {
3      public:
4          using Entry =  HashEntry< KeyType, DataType >; //!< Alias
5
6          HashTbl ( int _tblSize = DEFAULT_SIZE );
7          virtual ~HashTbl ();
8          bool insert ( const KeyType & _k, const DataType & _d ) throw ( std::bad_alloc );
9          bool remove ( const KeyType & _k );
10         bool retrieve ( const KeyType & _k, DataType & _d ) const;
11         void clear ( void );
12         bool empty ( void ) const;
13         unsigned long int count ( void ) const;
14         void showStructure () const;
15
16     private:
17         void rehash(); //!< Change Hash table size if load factor λ > 1.0
18         unsigned int mSize;  //!< Hash table size.
19         unsigned int mCount; //!< Number of elements currently stored in the table.
20         //!< The table: array of pointers to collision list.
```

```
21        std::forward_list< Entry > *mpDataTable;
22        // std::unique_ptr< std::forward_list< Entry > [] > mpDataTable;
23        //! Hash table's default size: 11 table entries.
24        static const short DEFAULT_SIZE = 11;
25 };
```

Os membros presentes na classe `HashTbl` têm a seguinte finalidade:

- `mSize`: Armazena o tamanho atual da tabela de dispersão, informado via argumento para o construtor. Se o cliente não informar um tamanho, uma tabela com 11 elementos é criada.

- `mCount`: Contagem atual de elementos armazenados na tabela. A inserção de novo elemento este membro é incrementado. Similarmente, a cada remoção bem sucedida de um elemento este membro deve ser decrementado.

- `*mpDataTable`: Ponteiro para receber a alocação dinâmica de um vetor de listas de entradas de tabela. A lista simplesmente encadeada do STL, `std::forward_list< Entry >`, ou a sua própria lista, `Forward_list< Entry >`, podem ser usadas para criar as listas de colisão associada a cada *endereço primário* da tabela. De maneira mais simples, cada elemento do vetor alocado `mpDataTable` é uma lista encadeada de itens de tabela. Se desejar, use *smart pointer* ou `std::unique_ptr< std::forward_list< Entry > [] > mpDataTable`.

Os métodos presentes na interface de `HashTbl` seguem a seguinte descrição:

- `HashTbl(int _tblSize)`: Construtor que aloca dinamicamente em `*mpDataTable` um vetor cujo tamanho é determinado como sendo o menor número primo maior do que o valor especificado em `_tblSize`. Esta é a área de armazenamento dos dados, ou seja, cada elemento deste vetor é um ponteiro para uma lista de itens de tabela, ou `HashEntry`.

- `~HashTbl()`: Destruidor da classe que libera a memória apontada por `mpDataTable`. Antes de liberar a memória, é importante percorrer o vetor liberando a memória utilizada por cada lista de colisão. Se usar *smart pointer*, este trabalho é feito por você!

- `insert(...)`: Insere na tabela a informação contida em `_d` e associada a uma chave `_k`. A classe calcula o endereço `end` que a informação `_d` deve ocupar na tabela e o armazena na lista de colisão correspondente, ou seja, em `mpDataTable[end]`. Se a inserção foi realizada com sucesso a função retorna `true`. Se a chave já existir na tabela, o método *sobrescreve* os dados da tabela com os dados contidos em `_d` e retorna `false`, para diferençar de uma inserção realizada pela primeira vez.

- `remove(const KeyType & _k)`: Remove um item de tabela identificado por sua chave `_k`. Se a chave for encontrada o método retorna `true`, caso contrário `false`.

- `retrieve(...)`: Recupera em `_d` a informação associada a chave `_k` passada como argumento para o método. Se a chave for encontrada o método retorna `true`, caso contrário `false`.

- `clear()`: Libera toda a memória associada às listas de colisão da tabela, removendo todos seus elementos. Note que a memória dinâmica associada ao ponteiro `mpDataTable` *não* deve ser liberada neste momento, mas apenas no destruidor da classe.
- `empty()`: Retorna `true` se a tabela de dispersão estiver vazia, ou `false` caso contrário.
- `count()`: Retorna a quantidade de elemento atualmente armazenados na tabela.
- `showStructure()`: É um método de *depuração* usado apenas para gerar uma representação textual da tabela e seus elementos. É um método útil na fase de desenvolvimento da classe para verificar se as operações sobre a tabela estão funcionando corretamente.
- `rehash()`: É um método de *privado* que deve ser chamado quando o fator de carga λ for maior que 1.0. O fator de carga é a razão entre o número de elementos na tabela e seu tamanho. Este método vai criar uma nova tabela cujo tamanho será igual ao menor número primo maior que o dobro do tamanho da tabela antes da chamada `rehash()`. Após a criação da nova tabela, todos os elementos devem ser inseridos na nova tabela, de acordo com uma nova função de dispersão secundária, baseada no novo tamanho da tabela. Cuidado especial deve ser tomado para evitar vazamento de memória. O cliente não deve perceber que esta operação foi acionada.

3.2 Definindo a Comunicação entre `HashTbl` e o Cliente

O cliente é responsável por passar para a tabela de dispersão quatro elementos: (i) O *tipo de informação* a ser armazenado na tabela; no nosso caso a classe `Account`; (ii) A *chave* única associada a uma informação; no nosso caso o tipo `AcctKey` definido dentro da classe `Account`. (iii) A *função de dispersão primária* que deve ser aplicada sobre a chave para se obter um *inteiro positivo* do tipo `std::size_t`. (iv) Uma forma de *comparar chaves por igualdade* de maneira que seja possível procurar por chaves iguais durante a busca linear executada sobre a lista de colisões, quando necessário.

Os últimos dois itens não foram ainda definidos explicitamente até o presente. O conceito e finalidade da função de dispersão primária já foram explicados na Seção 2, contudo. A *comparação de chaves por igualdade* é necessário para localizar, via busca linear, a chave na lista de colisões associada ao endereço primário. Tal busca linear, por sua vez, depende da comparação por igualdade entre a chave de entrada e cada uma das chaves já armazenadas na lista de colisão.

Como cada chave é definida pela aplicação cliente, cabe ao cliente também a responsabilidade de indicar para a classe `HashTbl` como ela deve proceder para verificar se duas chaves são *iguais*. Se uma chave for composta apenas de um tipo básico da linguagem, então uma comparação com `==` é suficiente. Por outro lado, se uma chave for composta de outros tipos, como o caso das chaves apresentas na **Versão 2** e **Versão 3** na Seção 1 poderá ser necessário realizar mais de uma comparação e combinar seus resultados por meio de alguma operação lógica. É possível fazer uso de funções de comparação padrão presentes no STL e definida para tipos básicos, `std::equal_to<T>`.

Bom, como tanto a função de dispersão primária como a função para comparar por igualdades duas chaves são códigos que dependem da aplicação cliente, elas devem ser passadas para a classe `HashTbl` de alguma maneira. Uma forma de passar *código* para uma função ou classe é por meio de *ponteiro para função*. Outra forma, mais conveniente, de passar código para uma classe é através de *argumentos templates* por meio de **function objects** ou "*functors*".

3.3 "Functors" para Passar Código para `HashTbl`

Um *functor* é uma classe simples que sobrecarrega o operador funcional `operator()`. Esta simples estratégia é muito flexível, visto que o operador funcional pode receber um número ilimitado de parâmetros e retornar qualquer tipo.

Para o caso da função de dispersão primária, precisamos de uma função que recebe como argumento uma chave, ou seja, `AcctKey` e retorna um inteiro sem sinal, ou seja, `std::size_t`. É importante (mas não obrigatório) retornar o tipo `std::size_t` para que a função de dispersão fique compatível com as funções de dispersão padrão do STL. Assim sendo, um functor para cumprir esta finalidade seria (chave do tipo **Versão 1**):

```
1 struct KeyHash {
2     std::size_t operator()(const Account::AcctKey& _k) const
3     {
4         return  std::hash<int>()( _k ) ;
5     }
6 };
```

Se a chave for a da **Versão 2**, ou seja, uma combinação de nome do cliente e número da conta, poderíamos ter o seguinte functor:

```
1 struct KeyHash {
2     std::size_t operator()(const Account::AcctKey& _k) const
3     {
4         return  std::hash<std::string>()( _k.first ) ) xor
5                 std::hash<int>()( _k.second ) ;
6     }
7 };
```

assumindo que a chave será armazenada como um `std::pair<std::string,int>`, onde o primeiro campo é o nome do cliente e o segundo o número da conta.

Para o caso da função de comparação de chaves, é necessário receber duas chaves e retornar um booleano indicando se elas são iguais, `true`, ou não, `false`. Portanto, poderíamos ter o seguinte functor para chaves da **Versão 1**:

```
1 struct KeyEqual {
2     bool operator()(const Account::AcctKey & lhs, const Account::AcctKey & rhs) const
3     {
4         return lhs == rhs;
5     }
6 };
```

Para chaves da **Versão 2** teríamos o seguinte functor:

```
struct KeyEqual {
    bool operator()(const Account::AcctKey& lhs, const Account::AcctKey& rhs) const
    {
        return lhs.first == rhs.first and lhs.second == rhs.second ;
    }
};
```

3.4 Ajustes Finais em `HashTbl`

Por fim, é necessário fazer uma alteração na definição original de `HashTbl` apresentada anteriormente na Seção 3.1 de maneira a incluirmos quatro parâmetros templates. Os novos argumentos templates estão destacados em vermelho.

```
template < typename KeyType,
           typename DataType,
           typename KeyHash = std::hash<KeyType>,
           typename KeyEqual = std::equal_to<KeyType> >
class HashTbl {
    public:
        //! It is used to make code more concise.
        using Entry = HashEntry< KeyType, DataType >;
    ...
};
```

Note que os quatro parâmetros template correspondem aos quatro elemento que o cliente precisa passar para a classe `HashTbl`, conforme discutido anteriormente na Seção 3.2. Em particular, os últimos dois argumentos template possuem valores *default*. Isso quer dizer que se o cliente quiser, pode passar apenas dois argumentos template. Os últimos dois serão mapeados, respectivamente, par a função de dispersão padrão do STL para o tipo de chave especificado e para uma função de comparação de igualdade padrão do STL para o tipo de chave especificado pelo cliente. Neste caso, se o tipo chave do cliente não corresponder a um tipo básico da linguagem um erro de compilação será gerado.

Para fazer uso dos argumentos template do tipo functor, basta instanciar o functor. Veja abaixo o exemplo de uso destes objetos dentro do método `insert`.

```
template < class KeyType, class DataType, class KeyHash, class KeyEqual >
bool HashTbl<KeyType, DataType,
             KeyHash, KeyEqual>::insert ( const KeyType & _Key,
                                          const DataType & _DataItem )
{
    KeyHash hashFunc;   // Instantiate the "functor" for primary hash.
    KeyEqual equalFunc; // Instantiate the "functor" for the equal to test.
    Entry newEntry ( _Key, _DataItem ); // Create a new entry based on arguments.
    // Apply double hashing method, one functor and the other with modulo function.
    auto end(  hashFunc( _Key ) % mSize  );
    ...
```

```
12          // Comparing keys inside the collision list.
13          if ( true == equalFunc( it.mKey, newEntry.mKey ) )
14          {
15              ...
16          }
17      ...
18  }
```

4 Considerações Finais

Sua tarefa consiste em desenvolver completamente a classe `HashTbl` e a aplicação para manipular constas correntes. Do lado do desenvolvedor de classes, procure testar amplamente os métodos de manipulação da tabela de dispersão descritos na Seção 3.1. Do lado do cliente, procure utilizar a tabela de dispersão para as três versões de chave sugeridas na Seção 1.

Para o caso da **Versão 3** em que uma chave deve ser composta por mais do que dois campos, sugere-se o uso de tuplas do STL, ou `std::tuple<class... Types>` para combinar os campos em um tipo `AcctKey`.

Neste caso será necessário alterar o método `getKey`, e os functors `KeyHash` e `KeyEqual`. Note, contudo, que *nenhuma alteração* precisa ser feita na classe `HashTbl`.

Por último, é importante destacar que o comportamento projetado para a classe `HashTbl` é similar (na verdade um subconjunto) do comportamento oferecido pela classe padrão do STL para o container associativo `std::unordered_map`.

◀ FIM ▶

Referências

[1] Jayme Luiz Szwarcfiter and Lilian Markenzon. *Estrutura de Dados e seus Algoritmos*, chapter 8 – Tabelas de Dispersão, pages 227–249. Livros Técnicos e Científicos Editora S.A., 1994.

[2] Mark Allen Weiss. *Data Structure and Algorithms in C++*, chapter 5 – Hashing, pages 193–244. Pearson Education, Inc. as Addison-Wesley, 2014.