Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • IMD0030
◁ Exercícios de Implementação de Dicionário com Vetor ▷
7 de junho de 2017

Apresentação

O objetivo deste exercício é oferecer uma oportunidade para implementar o Tipo Abstratos de Dado (TAD) **dicionário** com base em lista sequencial (i.e. vetor), ordenada e não ordenada.

Através deste exercício de implementação espera-se que os conceitos e operações sobre o TAD dicionário e a Estrutura de Dados (ED) lista sequencial, sejam consolidadas. Maiores detalhes teóricos sobre dicionários e listas sequenciais podem ser encontrados em livros de estruturas de dados como [2] ou [1].

Sumário

1 Introdução

Dicionário é um TAD que representa uma *coleção* que suporta armazenamento e recuperação de informação baseado em **conteúdo** (ou chave)—i.e. um *associative container*. Esta abordagem é diferente de outras TADs que permitem recuperação de conteúdo com base na especificação de **posição**, como é o caso de certos tipos de listas—i.e. um *sequence container*.

Um exemplo de dicionário implementado pela *Standard Template Library* (STL) do C++ é o `std::map`, enquanto que `std::vector` e `std::list` são exemplos de coleções cujo armazenamento e recuperação de informações é baseada em posição.

Vale ressaltar que uma TAD dicionário pode ser implementada através de vários tipos de EDs diferentes. É possível usar vetores (ordenados e não ordenados), listas encadeadas (ordenadas e não ordenadas), árvores *splay*, AVL, rubro-negra, bem como **tabela de dispersão**[1].

A escolha adequada da ED para implementar um dicionário vai depender das restrições impostas pela aplicação-fim e do desempenho (complexidade) temporal e espacial de cada operação de manipulação que uma ED oferece.

2 Operações

As operações *primária* que todo dicionário deve suportar são:

- `Insert(D, <k,i>)` ou `D.insert(k,i)`: Insere o par associado chave `k` e informação `i` no dicionário `D`.
- `Delete(D, k)` ou `D.delete(k)`: Remove (e opcionalmente pode retornar) do dicionário `D` a informação `i` associada à chave `x` .
- `Search(D,k)` ou `D.search(k)`: Busca e recupera do dicionário `D` a informação `i` associada à chave `k`.

Note que para um dicionário funcionar corretamente, é necessário que a chave associada a informação seja *única*, ou seja, não se deve armazenar informações com chaves duplicadas.

Alguns dicionários podem ocasionalmente suportar operações *secundárias* de certa utilidade, como por exemplo:

- `Max(D)` / `D.max()` ou `Min(D)` / `D.min()`: Recuperam do dicionário `D` a informação `i` com a maior (ou menor) chave, determinado por uma ordem estrita definida sobre a chave.
- `Predecessor(D,k)` / `D.predecessor(k)` ou `Sucessor(D,k)` / `D.sucessor(k)`: Recuperam do dicionário `D` a informação `i` cuja chave seja imediatamente anterior (ou posterior) a `k`, considerando uma ordem estrita definida sobre a chave.

3 Tarefa

Sua tarefa consiste em implementar dois tipos de dicionário com suporte às operações primárias e secundárias descritas na Seção 2. O primeiro deverá armazenar seus elementos internamente em um vetor de tamanho fixo (informado pelo cliente) sem manter as chaves ordenadas. O segundo deverá armazenar seus elementos internamente também em um vetor de tamanho fixo, porém seus elementos deverão ser mantidos ordenados de acordo com o valor da chave.

Para facilitar o processo de aprendizagem, recomenda-se o desenvolvimento do dicionário em etapas com nível crescente de complexidade, descritas a seguir.

[1] Nosso próximo assunto em EDB1.

3.1 Etapa #1: dicionário com vetor não ordenado

Nesta etapa você deve implementar uma classe representando um dicionário com armazenamento em vetor **não ordenado**, denominada de `DAL` (*Dictionary with Array List*). Assuma que a chave será do tipo inteiro (`int`) e que a informação será uma cadeia de caracteres (`std::string`).

Neste caso é possível contar com a ordem estrita já definida na linguagem `<` ou `std::less<int>()` [2] para comparar as chaves. Note que em uma etapa posterior (ver Seção 3.4) sua classe deverá receber a forma de comparação entre chaves como um argumento template, aumentando a flexibilidade de atuação da classe.

Confira a seguir uma sugestão inicial para a implementação da classe `DAL`.

```
1  class DAL {
2      protected:
3          using Key = int;             // Alias para Chave.
4          using Data = std::string;    // Alias para Dado.
5          struct NodeAL {              // Estrutura do nó, representando o par chave-informação.
6              Key id;                  // A chave é um inteiro simples.
7              Data info;               // A informação é uma cadeia de caracteres.
8          };
9
10         static const int SIZE=50;    // Tamanho default da lista.
11         int mi_Length;               // Comprimento atual da lista.
12         int mi_Capacity;             // Capacidade máxima de armazenamento.
13         NodeAL *mpt_Data;            // Área de armazenamento: vetor alocado dinamicamente.
14
15         int _search( const Key & _x ) const; // Método de busca auxiliar.
16
17     public:
18         DAL ( int _MaxSz = SIZE );
19         virtual ~DAL () { delete [] mpt_Data; };
20         bool remove( const Key & _x, Data & _s );  // Remove da lista.
21         bool search( const Key & _x, Data & _s ) const; // Busca publica.
22         bool insert( const Key & _newKey, const Data & _newInfo ); // Insere na lista.
23         Key min( ) const; // Recupera a menor chave do dicionário.
24         Key max( ) const; // Recupera a maior chave do dicionário.
25         // Recupera em _y a chave sucessora a _x, se existir (true).
26         bool sucessor( const Key & _x, Key & _y ) const;
27         // Recupera em _y a chave antecessora a _x, se existir (true).
28         bool predecessor( const Key & _x, Key & _y ) const;
29
30         //! Sends back to the output stream an ascii representation for the list.
31         inline friend
32         std::ostream &operator<< (std::ostream& _os, const DAL& _oList ) {
33             _os << "[ ";
34             for( int i(0) ; i < _oList.mi_Length ; ++i )
35                 _os << "{id: " << _oList.mpt_Data[i].id << ", info: "
36                     << _oList.mpt_Data[i].info << "}  ";
37             _os << "]";
38             return _os;
39         }
40 };
```

[2] Objecto função ou *functor* que executa comparações normalmente invocando o `operator<` sobre dois parâmetros, no caso inteiros.

3.2 Etapa #2: dicionário com vetor ordenado

Nesta etapa você deve implementar uma classe representando um dicionário em vetor **ordenado**, denominada de `DSAL` (*Dictionary with Sorted Array List*). Assuma que a chave será do tipo inteiro (`int`) e que a informação será uma cadeia de caracteres (`std::string`).

Novamente é possível contar com a ordem estrita já definida na linguagem `<` ou `std::less<int>()` para comparar as chaves.

Com objetivo de evitar a duplicação de código, faça com que `DSAL` seja uma **extensão** (herança em Orientação à Objetos) de `DAL`. Com isso, todo o arcabouço já existente na classe `DAL` será reaproveitado na nova classe.

Cabe a você, contudo, identificar quais métodos precisam ser *sobrescritos* ou seja, quais métodos possuem o mesmo nome mas devem apresentar comportamento diferente? Por exemplo, o método de busca interno `_search()` deve ser diferente para cada classe, visto que em `DAL` utilizamos uma busca linear simples, enquanto que em `DSAL` podemos usar uma busca binária para determinar o índice do vetor que contém a chave procurada.

Confira a seguir uma sugestão de implementação da classe `DSAL` com herança.

```
1  class DSAL : public DAL   // Indicação de herança.
2  {
3      public:
4          DSAL( int _MaxSz ) : DAL( _MaxSz ) { /* Empty */ };
5          virtual ~DSAL() { /* Empty */ };
6
7          // Métodos para sobrescrever.
8          bool remove( const Key & _x, Data & );
9          bool insert( const Key & _novaId, const Data & _novaInfo );
10         Key min( ) const; // Recupera a menor chave do dicionário.
11         Key max( ) const; // Recupera a maior chave do dicionário.
12         // Recupera em _y a chave sucessora a _x, se existir (true).
13         bool sucessor( const Key & _x, Key & _y ) const;
14         // Recupera em _y a chave antecessora a _x, se existir (true).
15         bool predecessor( const Key & _x, Key & _y ) const;
16
17     private:
18     int _search( const Key & _x ) const; // Método de busca auxiliar.
19 };
```

3.3 Etapa #3: classes template

Nesta etapa você deve modificar as duas classes anteriores de maneira que elas possam receber o tipo de informação como um argumento template. Desta maneira, o dicionário pode armazenar qualquer tipo de informação, desde que ela possa ser associada a uma chave inteira.

Como a chave continua sendo do tipo inteiro, ainda é possível contar com a ordem estrita já definida na linguagem `<` ou `std::less<int>()` para comparar as chaves.

Confira a seguir uma sugestão de implementação da classe `DAL` e `DSAL`. O código é idêntico ao anteriormente apresentado, sendo listado aqui apenas as diferenças associadas com a sintaxe com o uso de template.

```
template < typename Data >
class DAL {
    protected:
        using Key = int;            // Alias para Chave.
        ...
};

template < typename Data >
class DSAL : public DAL< Data >   // Indicação de herança.
{
    public:
        DSAL( int _MaxSz ) :
            DAL< Data >( _MaxSz ) { /* Empty */ };
        ...
};
```

Como desde o início foi utilizado um alias para o tipo de informação por meio de um `using Data = std::string;`, pouca mudança no código é necessária.

3.4 Etapa #4: template de chave e informação

Nesta última etapa você deve modificar as duas classes anteriores de maneira que elas possam receber tanto o tipo de informação quanto a chave como argumentos template. Desta maneira, o dicionário pode armazenar qualquer tipo de informação, associado a qualquer tipo de chave, tornando a TAD dicionário bem flexível.

Como a chave agora é um argumento template indicado pelo cliente da classe, é necessário também informar um mecanismos que o dicionário deve utilizar para comparar chaves e estabelecer uma ordem estrita. Note que a ordem estrita é necessária para as duas classes, visto que em ambas é necessário comparar chaves em busca de similaridade ou ordenação.

Confira a seguir uma sugestão de implementação da classe `DAL` e `DSAL`. O código é idêntico ao anteriormente apresentado, sendo listado aqui apenas as diferenças associadas com a sintaxe com o uso de template.

```
template < typename Key, typename Data, typename KeyComparator >
class DAL {
        ...
};

template < typename Key, typename Data, typename KeyComparator >
class DSAL : public DAL< Key, Data, KeyComparator > // herança.
{
    public:
        DSAL( int _MaxSz ) :
            DAL< Key, Data, KeyComparator >( _MaxSz ) { /* Empty */ };
        ...
};
```

4 Código Cliente

Considerando que o dicionário foi implementado com sucesso, resta apenas demonstrar como o cliente (ou aplicação) pode utilizar o dicionário para armazenar informações. Confira o código a seguir.

```
1  class MyKeyComparator {
2      public:
3          bool operator()( const int &lhs, const int &rhs ) const
4          {
5              return lhs < rhs;
6          }
7  };
8
9  int main ( ) {
10     //DAL< int, std::string, MyKeyComparator >  myList( 50 );
11     DSAL< int, std::string, MyKeyComparator  >  myList( 50 );
12
13     cout << ">>> Inserindo {2015003129, \"Jack\"}" << endl;
14     myList.insert( 2015003129, "Jack" );
15
16     cout << ">>> Inserindo {2014065190, \"John\"}" << endl;
17     myList.insert( 2014065190, "John" );
18 }
```

Note, em particular, as Linhas 1-7 que definem um *functor* e as linhas 10 e 11 que instanciam os dois tipos de dicionários solicitados. O dicionário `myList` armazena chaves inteiras, associadas a informações do tipo cadeia de caracteres e utiliza o *functor* `MyKeyComparator` para estabelecer uma ordem estrita entre as chaves.

O mecanismo que o C++ suporta para passarmos *código* para uma função ou classe é através de *ponteiro para função* ou *objecto função* (*functor*). No caso do dicionário, precisamos de uma função `bool compare(Key a, Key b)` que receba como argumento duas chaves, `a` e `b`, e retorne `true` se `a` $menordoque$ `b`, e `false` caso contrário.

O *functor* `MyKeyComparator` define como as chaves utilizadas pelo dicionário devem ser comparadas para estabelecer uma ordem estrita. No exemplo acima a comparação é simples, pois as chaves são do tipo inteiro.

Assim, se quisermos criar um novo dicionário cuja chaves sejam, digamos, cadeia de caracteres, tudo o que precisamos fazer é projetar um *functor* capaz de receber duas cadeias de caracteres e retornar `true` ou `false`, conforme a definição de `compare` fornecida anteriormente.

Vejamos mais um exemplo hipotético. Suponha que a chave seja um objeto da classe `Fruit` (fruta). É possível definirmos dois tipos de functors, por exemplo, caso desejemos comparar frutas por calorias—invocando `getCalories()` sobre cada fruta— ou por peso da fruta—invocando `getWeight()` sobre cada fruta. Veja abaixo o código correspondente a estes dois functors.

```
1  class FruitCaloriesComparator {
2      public:
3      bool operator()( const Fruit& lhs, const Fruit& rhs ) const {
4              // Assumindo que getCalories() retorna um inteiro.
5              return  lhs.getCalories() < rhs.getCalories();
6      }
7  };
```

```
8
9  class FruitWeightComparator {
10     public:
11     bool operator()( const Fruit& lhs, const Fruit& rhs ) const {
12             // Assumindo que getWeigth() retorna um inteiro.
13             return  lhs.getWeigth() < rhs.getWeigth();
14     }
15 };
```

◀ FIM ▶

Referências

[1] Jayme Luiz Szwarcfiter and Lilian Markenzon. *Estrutura de Dados e seus Algoritmos*, chapter 2 – Listas Lineares, pages 19–59. Livros Técnicos e Científicos Editora S.A., 1994.

[2] Mark Allen Weiss. *Data Structure and Algorithms in C++*, chapter 3 – Lists, Stacks, and Queues, pages 77–102. Pearson Education, Inc. as Addison-Wesley, 2014.