Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • IMD0030
◁ Exercícios de Implementação de Lista Encadeadas ▷
27 de abril de 2015

Objetivos

O objetivo deste exercício apresentar a estrutura de dados *lista duplamente encadeada*, enfatizando sua implementação através de classes em C++. A lista duplamente encadeada é usada no contexto da implementação de um **coleção linear de dados** com características de um (i) um *deque*, i.e. com suporte a inserções/remoções nas extremidades; bem como (ii) suporte a acesso—i.e. busca, inserção e remoção—randômico de elementos dentro da coleção.

A coleção, aqui denominada de `List`, é implementada por meio de uma lista duplamente encadeada. O exercício também envolve o conceito do *padrão de programação iterador* e sua relação com a coleção `List`, viabilizando operações de acesso sequencial e randômico sobre a coleção.

Após sua correta implementação, espera-se que a classe lista `List` possa ser utilizada, por exemplo, na implementação de outras TADs como de pilhas, filas e deques, bem como na solução de alguns problemas computacionais que requerem este tipo de coleção de dados.

1 Classe Lista Duplamente Encadeada

Nesta seção descrevemos os detalhes para a implementação de uma classe `List` com template. Esta classe implementa uma *lista duplamente encadeada* com nós *cabeça* e *calda*.

Antes de entrar em detalhes específicos da lista, faz-se necessário introduzir o conceito de **iteradores**, um padrão de programação (*design pattern* em Inglês) muito comum e importante para uma manipulação segura e eficiente de coleção de dados de maneira geral, e de listas em particular.

1.1 Iteradores

Iterador é um conceito importante relacionado a estruturas de dados que funcionam como *container* ou *coleção* de dados. Trata-se de um mecanismo semelhante a um apontador utilizado para acessar os elementos de um container de forma segura. De fato, o iterador é um *padrão de programação* adotado por bibliotecas profissionais como o STL (*Standard Template Library*) do C++ e a linguagem de progamação Java.

Em termos gerais a função da entidade iterador (implementada como uma classe em C++) é manter um apontador para uma certa posição do container, que no nosso caso será uma lista encadeada. Através de um objeto iterador é possível manipularmos ou acessar elementos da lista. Essa estratégia simples evita problemas básicos como "falha de segmentação", comumente provocada

por apontadores *inválidos* ou *selvagens*, i.e. apontadores que apontam para posições de memória desconhecidas, normalmente inválidas.

Apontadores inválidos são evitados ao se invocar métodos da classe iterador que verificam se o apontador é seguro, ou seja, se ele aponta para algum elemento válido do container. Além disso, a classe iterador pode oferecer sobrecarga de operadores, como por exemplo, `operator==` / `operator!=` para comparação de elementos. O uso de operadores relacionais facilitam, entre outras coisas, a operação de acessar sequencialmente—ou iterar—todos os elementos de um container.

Desta forma, a função do iterador é manter o *status* sobre qual é a posição da lista atualmente sendo acessada pelo código cliente. Para que essa relação "íntima" entre lista e iteradores dê certo é necessário que a classe iterador seja declarada como *amiga* (`friend`) da classe `List`, ou seja, a classe iterador terá privilégios de acesso à membros e métodos privados e/ou protegidos da classe `Lista`.

Confira o exemplo no Código 1 que demonstra o uso de iteradores associados à classe `std::vector` do STL. Este programa cria um objeto `vec1` do tipo `std::vector` que inicialmente contém os caracteres de uma string `s` (linha 11). Na linha 12 um segundo `std::vector`, `vec2`, vazio é criado. A linha 13 cria um iterador que vai percorrer um container do tipo `std::vector<char>`. Linhas 15–16 utilizam o iterador criado para percorrer o container `vec1` e inserir cada um de seus elementos em `vec2`, utilizando o método de `std::vector` chamado `push_back()`. Note que o iterador é utilizado na linha 16 para acessar um elemento do container `vec1`, usando a sobrecarga do operador `*` (em `*i`). Ao final, `vec2` possui uma cópia de `vec1`, fato testado na linha 18.

Código 1 Exemplo de uso de **iterador** com o container `std::vector` do STL.

```
1  #include <cassert>
2  #include <iostream>
3  using std::cout;
4  using std::endl;
5
6  #include <vector>
7  using std::vector;
8
9  int main ( ) {
10    const char *s = "Adoro iteradores!";
11    vector< char > vec1 ( &s[0], &s[strlen(s)] );
12    vector< char > vec2;
13    vector< char >::iterator i;
14
15    for ( i = vec1.begin(); i != vec1.end(); ++i )
16       vec2.push_back( *i );
17
18    assert ( vec1 == vec2 );
19    cout << " --- Ok." << endl;
20
21    return 0;
22 }
```

1.2 Classe `List`: Uma Lista Duplamente Encadeada

Esta classe deverá ser implementada como uma *lista duplamente encadeada*, sendo necessário manter dois apontadores para ambas extremidades da lista. Esta decisão permite um tempo de resposta constante, quando a operação é realizada sobre um ponto conhecido da lista (por exemplo, sobre um iterador ou sobre uma das extremidades da lista).

Para tanto serão necessários quatro classes, a saber:

- `List`, que conterá apontadores para ambas as extremidades da lista (o AIL `head` e o ACL `tail`), o tamanho atual da lista e mais um conjunto de métodos para *inserir*, *remover*, *procurar* e *consultar* os elementos da lista.
- `Node`, que será uma *estrutura aninhada privada*[1]. Um nó contém o dado d a ser armazenado, apontadores para os nós antecessor p e sucessor n no encadeamento, e um único construtor que recebe d, p e n como parâmetros. Como esta `struct` é definida dentro da classe `List`, os seus campos podem ser acessados diretamente com o operador '`.`', sem a necessidade de codificar métodos *accessors*/*mutators*.
- `const_iterator`, que abstrai a noção de posição, e é uma *classe aninhada pública*. O `const_iterator` é um *wrapper* que armazena um apontador para um determinado nó da lista, e porvê implementação das funcionalidades básicas de iteradores, todas na forma de *sobrecarga* de operadores tais como `=`, `==`, `!=`, `-`, `++`, `*`, etc..
- `iterator`, que abstrai a noção de posição, e é uma *classe aninhada pública*, derivada de `const_iterator` através de herança pública. O `iterator` tem a mesma funcionalidade do `const_iterator`, exceto pelo `operator*` que retorna uma referência para o item sendo visualizado, ao invés de uma referência *constante*. Note que `iterator` pode ser usado em qualquer rotina que requeira um `const_iterator`, mas o contrário não é verdade. Em outras palavras, um `iterator` *É-UM* `const_iterator`.

Para facilitar uma série de operações de manipulação da lista (por exemplo, evitando os casos especiais) e facilitar a implementação de iteradores, a classe `Lista` deverá possuir um nó-cabeça (denominado de `header`) e um nó-calda (denominado de `tail`). A Figura 1 apresenta uma ilustração esquemática da lista duplamente encadeada com os nós *sentinelas*.

Os Código 2 à 5 apresentam trechos com a declaração das classes, seus membros e todos os métodos que devem ser implementados. Perceba que todas as quatro classes são declaradas em um único arquivo denominado `List.h` e que o programa de teste (contendo o `main`) deve-se chamar de `driver_list.cpp`. Para melhor organizar seu código, crie as classes dentro do *namespace* `MyList`.

Implementação

Vamos descrever, brevemente a função dos métodos mais importantes da classe `List` apresentada no Código 2. Perceba que a lista possui dois nós especiais, `head` e `tail` (linhas 42 e 43), que

[1] Na verdade um `struct`, por *default*, é público, mas o `struct Node` deve ser declarado dentro da classe `List` sob a diretiva `private`.

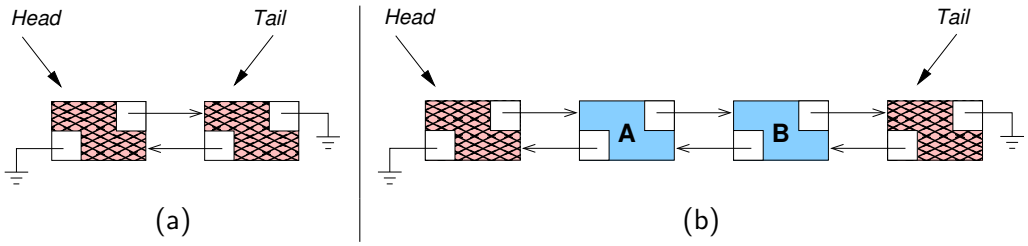

Figura 1: Exemplos de listas *duplamente encadeada* com *nó-cabeça* (`head`) e *nó-calda* (`tail`): (a) vazia e (b) com dois elementos.

devem ser alocados no construtor padrão e no construtor cópia, e devem ser desalocados apenas no destrutor para evitar vazamento de memória. Além da alocação destes dois nós, é preciso inicializar os outros campos da classe, como a quantidade de elementos e os apontadores de cada um dos nós especiais, conforme Figura 1-(a). Como estas ações devem ser realizadas duas vezes (no construtor e no construtor cópia), podemos centralizá-las no método privado `init()` (linha 46), bastando invocá-lo para acionar estas ações.

Antes de prosseguirmos, é importante oferecer uma explicação sobre *mutators* e *accessors*. Vários dos métodos de `List` apresentam estas duas versões. A única diferença entre as versões *mutator* e *accessor* de um mesmo método é que a primeira versão retorna um iterador normal (`iterador`) e a segunda um iterador constante (`const_iterator`) que não permite alteração do valor do nó apontado por ele.

Um dos grupos de métodos mais importantes são os `begin()` e `end()` . O `begin()` retorna um iterador para o **primeiro nó válido da lista**, ou seja o nó logo após o `head` . Já o método `end()` retorna um iterador para o **nó especial** `tail` , ao invés do último nó válido da lista. Esta sutil diferença é para viabilizar alguns idiomas de programação bem naturais em programas C++ e iteradores do STL, como

```
iterator itr = begin();
for ( ; itr != end(); ++itr ) // Busca elemento x na lista
    if ( *itr == x ) break;
```

neste caso se `end()` retornasse o último nó válido o laço ao invés do nó `tail` o trecho em vermelho não permitiria que o ultimo nó válido fosse testado no corpo do laço. Note também que caso o método `begin()` seja invocado em uma lista vazia, será retornado o nó `tail` (primeiro nó apontado por `head`), que é o comportamento correto, uma vez que o nó `tail` simboliza fim de fila.

Uma dica importante no desenvolvimento dos demais métodos é reutilizar ao máximo os métodos `begin()` e `end()` para percorrer a lista e acessar elementos. O próximo grupo de métodos são os de operação básica como inserção e consulta na frente e na calda da fila (linhas 26 à 33), comportamento semelhante ao da estrutura de dados *Deque*. Note que os métodos `front()` e `push_front()` já estão implementados! Os métodos `size()` , `empty()` e `clear()` são auto-explicativos, com especial atenção para o método `clear()` que deve percorrer a lista para desalocar cada um de seus nós de dados, mas sem remover os nós cabeça e calda.

Código 2 Listagem parcial da classe `List`. Esta listagem contém referências para Códigos 3, 4 e 5, que contém a listagem das classes relacionadas.

```
1  template <typename Object>
2  class List {
3    private:
4      // O nó básico da lista.
5      struct Node { Veja Código 3 };
6
7    public:
8      // Classes iteradores.
9      class const_iterator { Veja Código 4 };
10     class iterator : public const_iterator { Veja Código 5 };
11
12     // Métodos básico que todas as classes deveriam oferecer
13     List( );
14     ~List( );
15     List( const List & rhs );
16     const List & operator= ( const List & rhs );
17
18     // Métodos específicos da classe
19     iterator begin( ); // Versão mutator
20     const_iterator begin( ) const; // Versão accessor
21     iterator end( ); // Versão mutator
22     const_iterator end( ) const; // Versão accessor
23     int size( ) const; // Retorna tamanho da lista
24     bool empty( ) const; // Retorna true se vazia, falso caso contrário
25     void clear( ); // Apaga todos os nós da fila, tornando-a vazia
26     Object & front( ) { return *begin( ); }
27     const Object & front( ) const;
28     Object & back( );
29     const Object & back( ) const;
30     void push_front( const Object & x ) { insert( begin( ), x ); }
31     void push_back( const Object & x );
32     void pop_front( );
33     void pop_back( );
34     iterator insert( iterator itr, const Object & x );
35     iterator erase( iterator itr ); // remove nó apontado por itr
36     iterator erase( iterator start, iterator end );
37     const_iterator find( const Object & x ) const;
38     iterator find( const Object & x );
39
40   private:
41     int    theSize;
42     Node *head;
43     Node *tail;
44
45     // Inicializa campos para representar lista vazia
46     void init( );
47 }; // Classe List
48
49 #endif
```

Logo a seguir, nas linhas 34 à 36, temos os métodos para inserção/remoção de elementos na/da

Código 3 Listagem parcial da estrutura `Node`, parte da classe `List` (confira Código 2).

```
1  // O nó básico da lista duplamente encadeada. Aninhada dentro de
2  // List, a estrutura 'Node' é pública (dentro de List), mas é ainda
3  // assim é encapsulada para cliente pois é declarada na categoria
4  // 'private' em List.
5  struct Node {
6      Object  data; // Campo de dados
7      Node   *prev; // Apontador para o próximo nó
8      Node   *next; // Apontador para o nó anterior
9
10     // Construtor inline com vários parâmetro default
11     Node( const Object& d = Object( ), Node* p = nullptr, Node* n = nullptr )
12       : data( d ), prev( p ), next( n ) { /* Empty */ }
13 };
```

lista. Este métodos trabalham sobre iteradores passados como argumentos, em especial o `insert()` insere um elemento **antes** o nó apontado pelo iterador passado como argumento, e o `erase()` com dois parâmetros — `start` e `end` — que remove todos nos na faixa $[start; end)$, ou seja, nó apontado pelo parâmetro `end` não é removido (intervalo fechado-aberto).

Por último temos o método `find()` que busca um elemento na lista, retornando o iterador para o nó em questão se ele estiver na lista ou `end()` (final de fila) caso o elemento não seja encontrado.

Implementação dos Iteradores

Os métodos das classes de iteradores são bem auto-explicativos e dizem respeito a manipulação de um apontador (`current`, declarado na linha 23 do Código 4, página 7) que aponta para um nó da lista. As operações básica são de acesso do valor apontado por `current` com o operador '`*`' e as operações de avanço e retrocesso (`operator++` e `operator--`).

Para diferenciar a chamada `itr++` de `++itr` são declarados, respectivamente, os operadores `operator++()` e `operator++(int)`. O parâmetro `int` é usado apenas para diferenciar entre os dois métodos (assinatura diferentes) — o mesmo vale para o operador de decremento '`--`'. Vale ressaltar que o operador de pré-incremento (`++itr`) e pré-decremento (`--itr`) são mais eficientes do que suas contrapartidas em pós-incremento e pós-decremento (i.e. `itr++` e `itr--`). Você saberia justificar o porque disto?

Como `iterator` é declarado como uma extensão de `const_iterator`, alguns de seus métodos podem ser re-aproveitados, como o operador de comparação (`==`) e diferença (`!=`).

◀ FIM ▶

Código 4 Listagem da classe `const_iterator`, parte da classe `List` (confira Código 2).

```
1  class const_iterator {
2    public:
3      // Construtor público
4      const_iterator( );
5      // Retorna objeto armazenado na posição atual.
6      // Para const_iterator, este método é um accessor que retorna
7      // uma referência constante. Logo este operador só pode
8      // aparecer do lado direito de uma atribuição ou em comparações.
9      const Object & operator* ( ) const;
10     // Pré-incremento: ++it
11     const_iterator & operator++ ( );
12     // Pós-incremento: it++
13     const_iterator operator++ ( int );
14     // Pré-decremento: --it
15     const_iterator & operator-- ( )
16     // Pós-decremento: it--
17     const_iterator operator-- ( int )
18     bool operator== ( const const_iterator & rhs ) const;
19     bool operator!= ( const const_iterator & rhs ) const;
20
21   protected:
22     // Declarado como protected para ser acessível pela classe 'iterator'
23     Node *current;
24     // Construtor protegido que recebe um nó para ser apontado.
25     // Utilizado dentro da classe 'List' apenas e não pelo código cliente.
26     const_iterator( Node* p );
27     // Necessário p/ permitir acesso de 'List' aos campos desta classe.
28     friend class List<Object>;
29 };
```

Código 5 Listagem da classe `iterator`, parte da classe `List` (confira Código 2).

```
1  // inheritance: IS-A relation
2  class iterator : public const_iterator {
3    public:
4      // Construtor público do iterator que invoca construtor da classe base.
5      iterator( ) { /* Empty */ }
6      // Retorna objeto armazenado na posição apontada por 'current'.
7      // For iterator, tem duas versões, uma accessor que permite sua
8      // utilização do apenas do lado direito de uma atribuição ou em
9      // comparações.
10     const Object & operator* ( ) const;
11     // Esta versão mutator é usada do lado esquerdo de atribuições.
12     Object & operator* ( );
13
14     // prefixo
15     iterator & operator++ ( );
16     // posfix
17     iterator operator++ ( int );
18     // prefixo
19     iterator & operator-- ( );
20     // posfix
21     iterator operator-- ( int );
22
23   protected:
24     // Construtor protegido que espera uma posição para apontar.
25     // É invocado principalmente dentro da classe 'List', mas
26     // não pelo cliente (que não tem acesso a este método).
27     iterator( Node *p );
28
29     // Necessário p/ permitir acesso de 'List' aos campos desta classe.
30     friend class List<Object>;
31 };
```