Apontamentos de Estruturas de Dados

Listas

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Outubro de 2022

Índice

8. Listas

8.1. Introdução

As listas são coleções simples ordenadas linearmente. Algumas coisas às quais nos referimos na vida quotidiana como listas, por exemplo listas de compras, são na realidade conjuntos porque a ordem não tem importância. A ordem tem importância nas listas. Uma lista de tarefas por exemplo é realmente uma lista se as tarefas a serem feitas estão na ordem em que devem ser concluídas (ou em alguma outra ordem).

> **Lista:** Uma coleção ordenada.

Como a ordem é importante nas listas devemos especificar uma localização ou índice de elementos quando modificamos a lista. Os índices podem começar em qualquer número mas iremos usar a convenção e atribuímos ao primeiro elemento de uma lista o índice `0`, o segundo índice `1` e assim por diante.

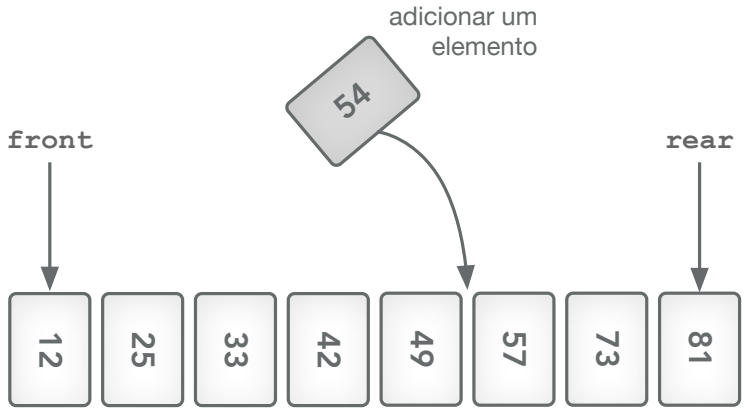
"Ordenado" nesta definição significa que cada elemento tem uma posição na lista. Portanto, o termo "ordenado" neste contexto não significa que os elementos da lista sejam ordenados por valor. Isto quer dizer que apesar da lista ser sempre ordenada temos da mesma forma três tipos de coleções: listas ordenadas, listas não ordenadas e listas indexadas.

Listas Ordenadas

Os elementos numa lista ordenada estão ordenados por alguma característica intrínseca dos seus elementos como por exemplo:

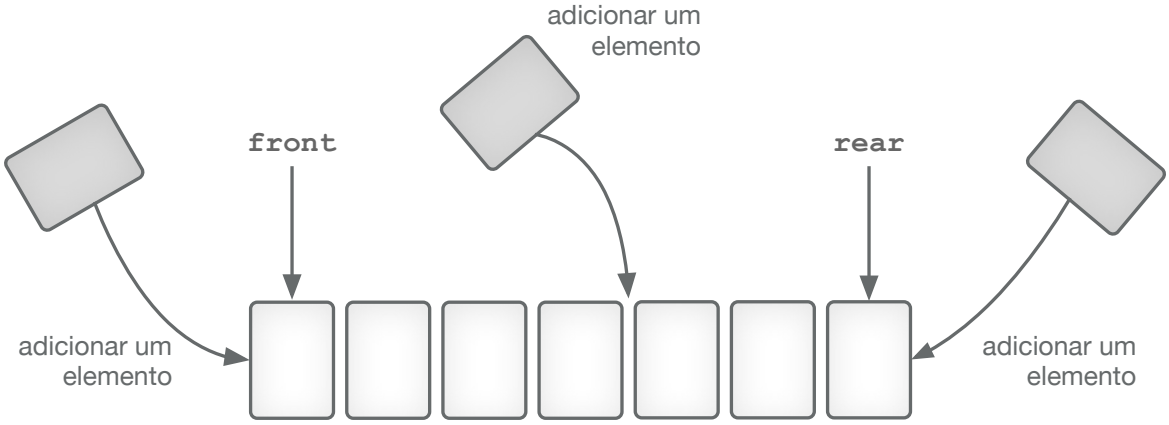
- nomes por ordem alfabética;
- valores por ordem ascendente.

Portanto, terão de ser os próprios elementos a determinar onde são armazenados na lista!



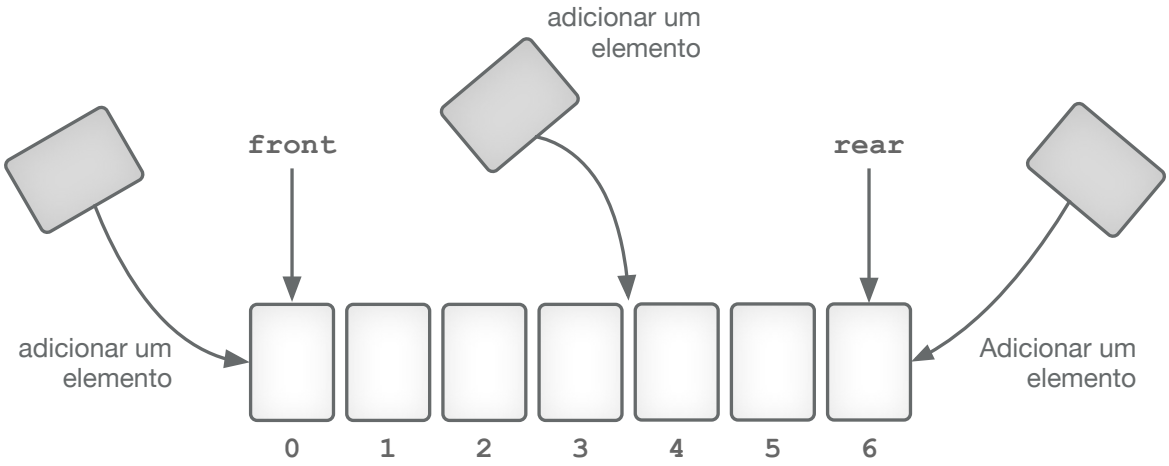
Listas Não Ordenadas

Tal como já foi referido anteriormente, os elementos numa lista não ordenada também têm uma ordem, mas esta não é baseada em qualquer característica do elemento a ser adicionado. O utilizador da lista determina a ordem dos elementos à medida que os vai adicionando. Um novo elemento pode ser colocado na parte dianteira ou traseira da lista, ou pode ser inserido após um determinado elemento existente na lista.



Listas Indexadas

Numa lista indexada, os elementos são referenciados pela sua posição numérica na lista. Como numa lista não ordenada não existe uma relação intrínseca entre os elementos o utilizador pode determinar a ordem. De cada vez que a lista é alterada os índices são atualizados.



8.2. List ADT

As listas são coleções de valores de algum tipo, então o ADT é lista de `T`, onde `T` é o tipo dos elementos na lista. O conjunto de valores deste tipo é o conjunto de todas as sequências ou tuplos ordenados de elementos do tipo `T`. O conjunto de valores inclui assim a lista vazia, as listas com um elemento do tipo `T`, as listas com dois elementos do tipo `T` (pares ordenados), as listas com três elementos do tipo `T` (triplos ordenados) e assim por diante. Portanto, o conjunto de valores deste ADT é o conjunto de todos os tuplos do tipo `T`, incluindo o tuplo vazio. Existem muitas operações que podem ser incluídas numa `List ADT`:

Operação	**Descrição**
`removeFirst`	Remove o primeiro elemento da lista
`removeLast`	Remove o último elemento da lista
`remove`	Remove um determinado elemento da lista
`first`	Examina o elemento na frente da lista
`last`	Examina o elemento na traseira da lista
`contains`	Determina se a lista contém um determinado elemento
`isEmpty`	Determina se a lista está vazia
`size`	Determina o número de elementos da lista
`iterator`	Retorna um iterador para os elementos da lista
`toString`	Representação da lista em string

Existem muitas operações comuns aos três tipos de lista que vimos anteriormente, estas incluem a remoção de elementos de várias maneiras assim como outras operações como verificar o *status* da lista. As principais diferenças entre os tipos da lista envolvem a forma como são adicionados os elementos. Por esse motivo não é possível descrever na interface de topo relativa à lista esse tipo de operações.

No caso das listas ordenadas fica a faltar apenas uma operação para adicionar um elemento.

Operação	Descrição
`add`	Adicionar um elemento à lista

Já no caso das listas não ordenadas como a localização do elemento é importante para o significado da lista temos três operações.

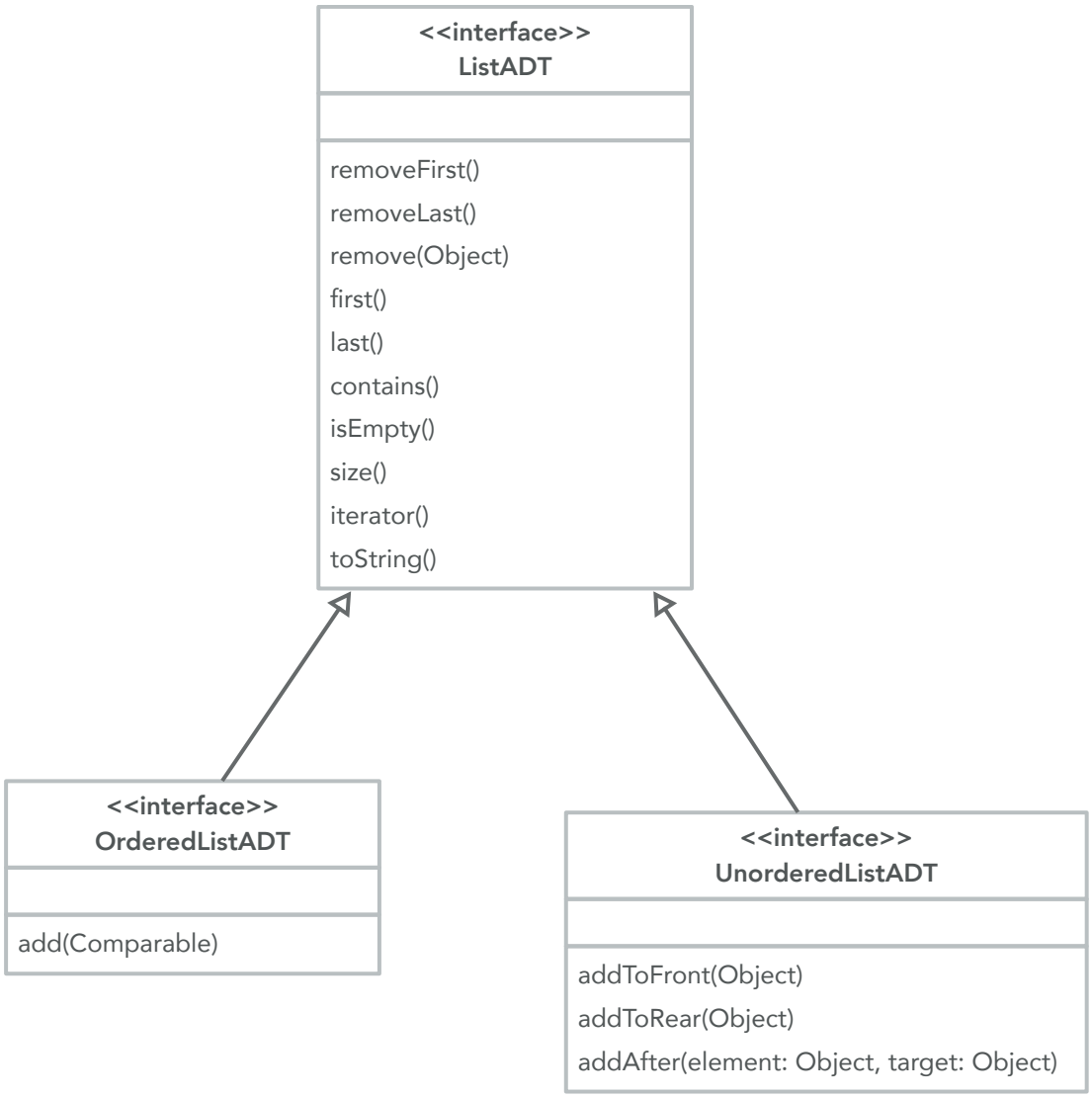
Operação	Descrição
`addToFront`	Adiciona um elemento à frente da lista
`addToRear`	Adiciona um elemento à traseira da lista
`addAfter`	Adiciona um elemento depois de um outro existente na lista

Tal como acontece com outras coleções, são usadas interfaces Java para definir as operações de coleção. Lembrem-se que as interfaces podem ser definidas através de herança (derivado de outras interfaces). É definida a lista comum de operações numa única interface depois são derivadas outras interfaces para os outros tipos de lista mais específicos.

8.3. Interface List

Fiel à noção de um ADT, uma interface não especifica como as operações são implementadas. Tal como fizemos anteriormente iremos discutir a implementação de uma lista através de *array* e de lista ligada. Existe, no entanto, uma grande diferença na implementação da lista versus as implementações de pilhas ou filas. No caso da lista temos pelo menos dois tipos diferentes de implementação: a lista em que as características dos elementos são fundamentais para a definição da ordem; e a lista em que essa informação não é importante.

Nesse sentido a figura seguinte apresenta um diagrama que descreve as interfaces dos ADTs a serem implementados e como se organizam.



Como pode verificar a partir da figura anterior a organização de interfaces para o nosso tipo é bastante diferente do que temos vindo a fazer. A lista é um tipo mais abstrato do que a pilha ou a fila portanto é possível serem usadas em mais contextos tornado-a uma coleção muito mais versátil. Não nos podemos esquecer que essa versatilidade vai também acompanhada por uma maior complexidade na implementação. Vão ser apresentadas de seguida as várias interfaces que constituem os ADTs e mais adiantes serão também dados alguns detalhes para as implementações baseadas em *array* ou lista ligada. A interface `ListADT<T>` sendo a mais geral (mais abstrata) tem todos os comportamentos comuns às listas:

```
1. import java.util.Iterator;
2.
3. public interface ListADT<T> extends Iterable<T> {
4.    /**
5.     * Removes and returns the first element from this list.
6.     *
7.     * @return  the first element from this list
8.    T removeFirst ();
9.    /**
10.    * Removes and returns the last element from this list.
11.    *
12.    * @return  the last element from this list
13.    */
14.   T removeLast ();
15.   /**
16.    * Removes and returns the specified element from this list.
17.    *
18.    * @param element  the element to be removed from the list
19.    */
20.   T remove (T element);
21.   /**
22.    * Returns a reference to the first element in this list.
23.    * @return  a reference to the first element in this list
24.    */
25.   T first ();
26.   /**
27.    * Returns a reference to the last element in this list.
28.    * @return  a reference to the last element in this list
29.    */
30.   T last ();
31.   /**
32.    * Returns true if this list contains the specified target
33.    * element.
34.    * @param target  the target that is being sought in the list
35.    * @return        true if the list contains this element
36.    */
37.   boolean contains (T target);
38.   /**
```

```
39.     * Returns true if this list contains no elements.
40.     * @return  true if this list contains no elements
41.     */
42.    boolean isEmpty();
43.    /**
44.     * Returns the number of elements in this list.
45.     *
46.     * @return  the integer representation of number of
47.     * elements in this list
48.     */
49.    int size();
50.    /**
51.     * Returns an iterator for the elements in this list.
52.     *
53.     * @return  an iterator over the elements in this list
54.     */
55.    Iterator<T> iterator();
56.    /**
57.     * Returns a string representation of this list.
58.     *
59.     * @return  a string representation of this list
60.     */
61.    @Override
62.    String toString();
63.}
```

A interface `OrderedListADT<T>` tem apenas um método o método `add(T element)` já que se trata de uma lista ordenada portanto não existe mais que uma forma de adicionar elementos.

```
1. public interface OrderedListADT<T> extends ListADT<T> {
2.     /**
3.      * Adds the specified element to this list at
4.      * the proper location
5.      *
6.      * @param element  the element to be added to this list
```

```
7.      */
8.     void add(T element);
9.  }
```

Por último temos a interface `UnorderedListADT<T>` que possui três forma distintas de adicionar elementos. O objetivo é ser bastante versátil e dar várias opções para quem a for utilizar.

```
10.public interface UnorderedListADT<T> extends ListADT<T> {
11.    /**
12.     * Adds the specified element to this list at
13.     * the proper location
14.     *
15.     * @param element  the element to be added to this list
16.     */
17.    void addToFront(T element);
18.    void addToRear(T element);
19.    void addAfter(T element);
20.}
```

Ao contrário das coleções implementadas até agora, a lista pode ser percorrida, isto é, os elementos podem ser pesquisados. Por esse motivo, que pode passar despercebido à primeira vista, é implementada a interface `Iterable<T>`. A obrigação de implementação do método `iterator()` é uma consequência do contrato `Iterable<T>` - são dados mais detalhes na secção **8.7. Padrão de Software Iterator**.

8.4. Usar Listas – Um Exemplo

Suponha que um programa de calendário tem uma lista de tarefas cujos elementos são ordenados por precedência de modo que o primeiro elemento da lista deve ser feito primeiro, o segundo depois e assim por diante. Os itens da lista de tarefas são todos visíveis num painel com uma barra de *scroll*; os itens podem ser adicionados, removidos ou movidos livremente na lista. Ao passar o rato sobre os itens da lista exibe detalhes sobre o item e que podem ser

alterados. Os utilizadores podem pedir para ver ou imprimir sublistas (como os últimos dez itens da lista), ou podem solicitar detalhes sobre um item da lista com uma certa precedência (como por exemplo o quinto elemento).

Claramente, uma lista é o tipo certo de *container* para listas de tarefas. A iteração sobre uma lista para exibir o seu conteúdo é fácil com um iterador interno ou externo. As operações de adicionar e remover permitem que os itens sejam inseridos na lista, removidos ou simplesmente movidos de posição.

8.5. Implementar uma List ADT com recurso a um *array*

As listas são muito fáceis de implementar com *arrays*. Os *arrays* fixos obrigam a que haja um tamanho máximo de lista enquanto os *arrays* dinâmicos permitem que as listas cresçam sem limite. A implementação é semelhante em ambos os casos. Um *array* é alocado para conter o conteúdo da lista com os elementos colocados no *array*. Um contador mantém o tamanho atual da lista. Se adicionarmos elementos num determinado local é necessário que os elementos seguintes sejam deslocados de forma a ser criado espaço para o novo elemento. Quando um elemento é removido temos que fazer o inverso, deslocar para trás uma posição para não ficarmos com um espaço vazio no *array*.

Os *arrays* fixos têm o seu tamanho definido em tempo de compilação, portanto, uma implementação com recurso a um array fixo não pode acomodar listas de tamanho arbitrário. Por outro lado, uma implementação com *array* dinâmico pode alocar um *array* maior e se a lista exceder a capacidade do *array* atual durante a execução é realocado. Realocar o *array* é uma operação pesada porque deve ser criado um *array* maior e o conteúdo do *array* antigo deve ser copiado para o novo *array*. Para evitar este custo, o número de realocações de *array* deve ser mínimo. Uma abordagem popular é duplicar o tamanho do *array* sempre que for necessário mais espaço. Por exemplo, vamos supor que uma lista começa com uma capacidade de `10`. À medida que se expande a capacidade é alterada para `20`, depois para `40`, depois para `80` e assim por diante. O *array* nunca se torna menor.

8.6. Implementar uma List ADT com recurso a uma lista ligada

A implementação ligada da `ListADT` usa uma estrutura de dados ligada para representar os valores do conjunto. Uma lista simplesmente ou duplamente ligada pode ser usada, dependendo das necessidades dos clientes. Vamos considerar o uso de listas simplemente ou duplamente ligadas para ilustrar as alternativas de implementação.

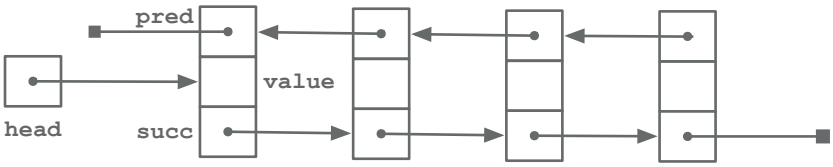
Considere a utilização de uma lista simplesmente ligada para armazenar um conjunto de elementos. A lista consiste numa referência, tradicionalmente denominada de cabeça que tem o valor `null` quando a lista está vazia e uma referência para o primeiro nó da lista caso contrário. O comprimento da lista também é normalmente guardado numa variável (não é obrigatório).

As operações da lista não necessitam de qualquer índice como argumento, no caso de pretender fazer alguma funcionalidade que necessite do índice ou elemento terá de percorrer a lista para localizar o nó na posição `i` ou na posição `i-1`. Porquê `i-1`? Porque tanto para a adição como para a remoção o campo de ligação no nó anterior ao nó `i` terá que ser alterado. De qualquer forma, se as listas forem longas e existirem muitas mudanças no final da lista terá de ser feito muito processamento para encontrar as posições para fazer as alterações.

Pode ser útil manter uma referência para o final da lista. Dessa forma as operações no final da lista podem ser feitas rapidamente em troca de um pequeno esforço extra de manter a referência extra. Se um cliente fizer muitas operações no final de uma lista o trabalho extra será justificado.

Outra maneira de tornar as operações da lista mais rápidas é armazenar os elementos numa lista duplamente ligada na qual cada nó (exceto aqueles nas extremidades) tem uma referência tanto para o seu sucessor como para o seu predecessor. A figura seguinte ilustra essa configuração. Usar um cursor com uma lista duplamente ligada pode acelerar consideravelmente as coisas porque os *links* possibilitam mover tanto para trás como para a frente da lista. Se uma operação precisa chegar ao nó `i` e o cursor marca o nó `j`, que está mais próximo do nó `i` do que o nó `i` está no início da lista, seguir os *links* a partir do cursor

permite chegar ao nó `i` mais rapidamente do que seguir os *links* a partir da cabeça da lista. Manter uma referência do final da lista torna as coisas ainda mais rápidas: é possível começar a seguir em direção a um nó a partir de três pontos: frente da lista, o final da lista ou o cursor, que geralmente está algures a meio da lista.



Existe ainda uma outra abordagem que não será explicada aqui que consiste em tornar a lista circular. De forma muito resumida para o fazermos `pred` do primeiro nó tem de apontar para o último nó em vez de conter `null` e `succ` do último nó tem de apontar para o primeiro nó em vez de conter `null`. Dessa forma não existe necessidade de uma referência extra para o final da lista.

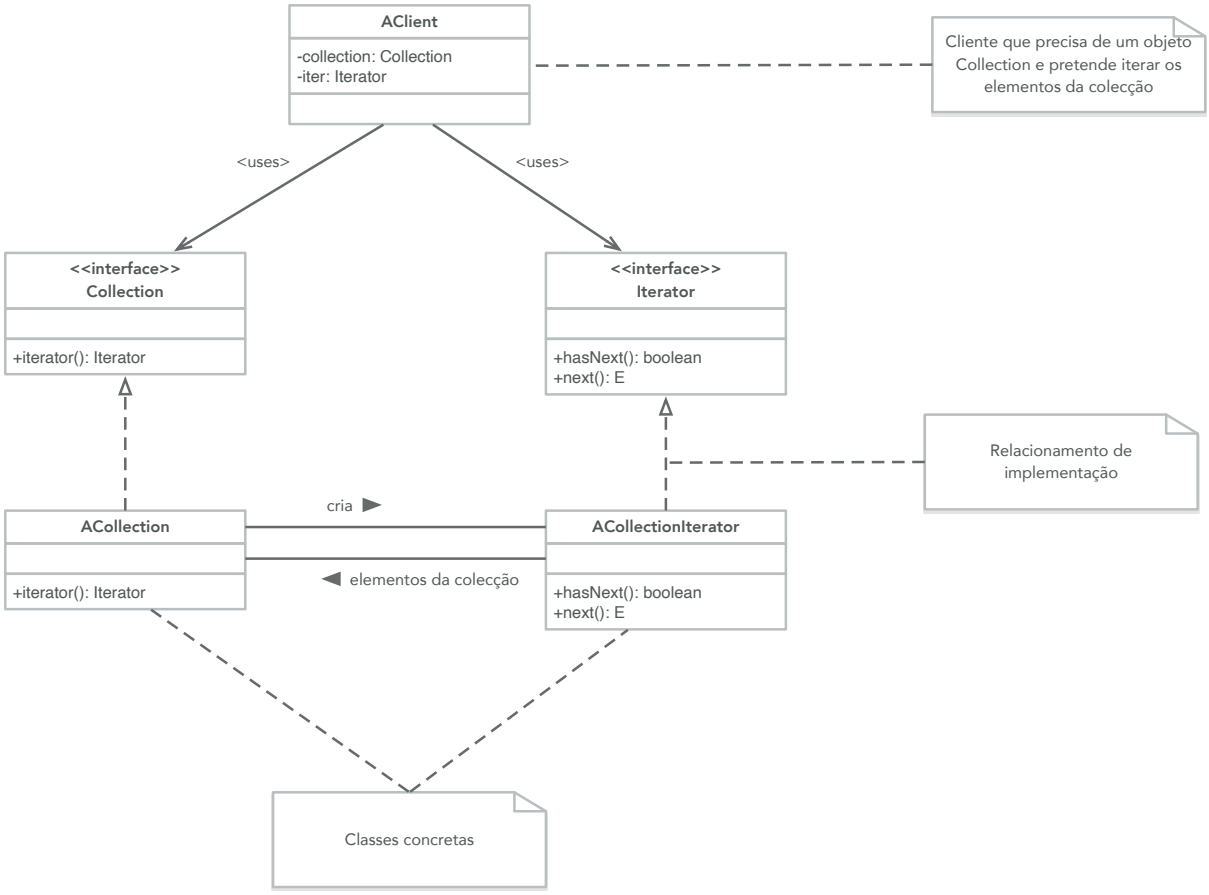
Modificar implementações em lista ligada é muito rápido uma vez que os nós nos quais se pretende operar sejam encontrados. Usar listas duplamente ligadas com cursores pode tornar a localização de nós bastante rápida. Uma implementação ligada é uma boa escolha quando é necessária uma grande variedade de operações de lista.

8.7. Padrão de Software Iterator

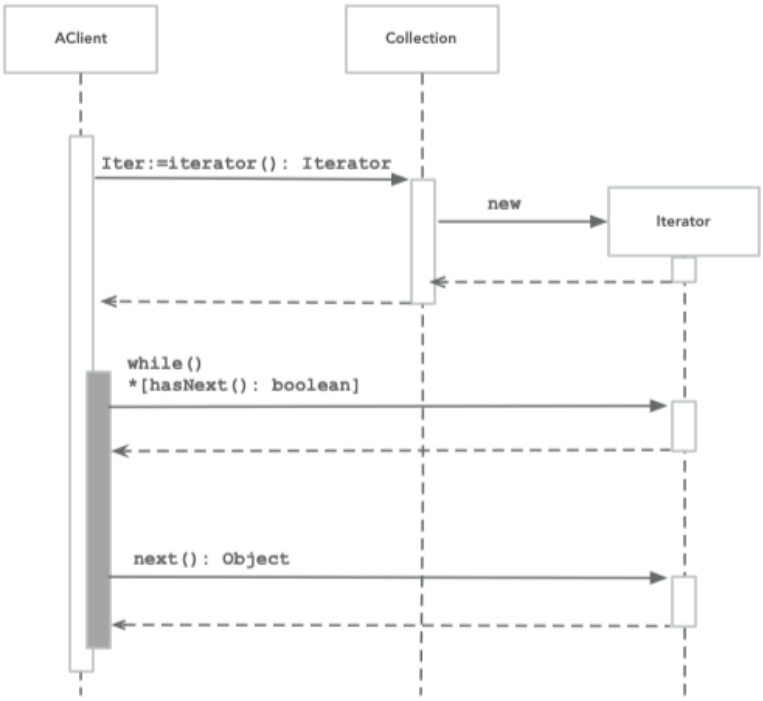
Problema: A necessidade de percorrer os elementos de uma coleção de uma forma comum para todos os tipos de coleções.

Solução: Fornecer uma interface `iterator` (iterador) que regule como os iteradores se irão comportar e obrigar a que um objeto da coleção forneça um objeto `iterator` capaz de percorrer todos os elementos da coleção.

De seguida é apresentado um diagrama de classes que demonstra a implementação e utilização do padrão de software *iterator*.



De seguida é apresentado o diagrama de sequência acerca da utilização de um iterador. É possível verificar quais os passos que sucedem desde o pedido à iteração propriamente dita.



1. O cliente pede ao objeto `collection` pelo objeto `iterator`;
2. O cliente usa o objeto `iterator` para percorrer os elementos da `collection`.

Atenção: Podem existir múltiplos iteradores simultaneamente ativos numa coleção.

Preste atenção à interface `Iterator`.

Métodos Obrigatórios	
`boolean hasNext()`	Retorna verdadeiro se a iteração tiver mais elementos
`E next()`	Retorna o elemento seguinte da iteração
Métodos Opcionais Todos os métodos opcionais não totalmente suportados por uma classe de implementação devem lançar um `UnsupportedOperationException`.	
`void remove()`	Remove a coleção subjacente que o elemento retornou na última chamada ao `next()`. Uma vez que o elemento foi removido, `remove()` não pode ser chamado novamente até que outra chamada para `next()` seja feita.

Dois dos métodos são obrigatórios: `boolean hasNext()` e `E next()`. O remove é um método opcional, no entanto como em Java não é possível definir métodos opcionais lançamos uma exceção do tipo `UnsupportedOperationException` para informarmos o cliente que a operação não está implementada.

Para a implementação de um iterador sugere-se a criação de uma classe aninhada na lista. Desta forma podemos ter acesso aos elementos da coleção já que o iterador precisa ter conhecimento interno da implementação da coleção, neste caso da lista. Assim mapear o modelo lógico de um iterador para a estrutura que armazena os elementos da coleção.

Modificações Concorrentes

O que acontece se a coleção for modificada enquanto uma iteração está em curso? Este problema é denominado de modificação concorrente e ocorre quando a coleção é modificada estruturalmente enquanto uma interação está em curso. Alguns cenários que podem suceder:

- Um elemento é adicionado à coleção e é colocado antes da posição do cursor da sequência de iteração em curso - neste caso a alteração será passada despercebida;
- O último elemento da sequência de iteração é removido de forma a que o cursor está agora na ultima posição, talvez confundindo o método `hasNext()`
    - throw a `ConcurrentModificationException`

Modificações Concorrentes: dois problemas

- **Problema 1:** Detetar modificação concorrente

  Solução: adicionar uma variável `modcount` à coleção para manter um registo das modificações estruturais (*add*/*delete*)
    - `modcount` é incrementado cada vez que o objeto da coleção é estruturalmente modificado a partir dos métodos `add` ou `remove`;
    - Quando um objeto `BasicIterator` é criado copia o valor da variável `modcount` da coleção para uma variável local `expectedModCount`;
    - Os métodos do iterador procuram por modificações concorrentes verificando se a variável `expectedModCount` é igual a `modcount`;

- Se não existirem alterações na estrutura da coleção as duas variáveis serão iguais caso contrário será lançada uma exceção.

- **Problema 2:** `Iterator.remove()` tem duas cláusulas
  - o elemento a remover é o ultimo elemento retornado pelo `next();`
  - `Iterator.remove()` apenas pode ser invocado uma vez por chamada ao `next()` (já que o `next()` fornece o elemento ao remove).

  Solução: O problema pode ser resolvido com recurso a uma *flag* `okToRemove` na classe `BasicIterator` que é
  - Inicializada a `false`;
  - Colocada a `true` pelo `next();`
  - Colocada a `false` pelo `Iterator.remove()`.

  Desta forma irá garantir que cada chamada ao `remove()` estará coordenada com o método `next()`.

8.8. Notas Finais

A implementação contígua de listas é muito fácil de programar e bastante eficiente para acesso a elementos mas bastante lenta para a adição e remoção de elementos. A implementação em lista ligada é consideravelmente mais complexa de programar e pode ser lenta se as operações necessitarem percorrer a lista a partir da cabeça. Se for usada uma lista duplamente ligada assim como um cursor, as operações da lista podem ser bastante rápidas em todos os aspetos. A implementação contígua é preferível para listas que não mudam com frequência e que têm de ser acedidas rapidamente enquanto a implementação ligada é melhor quando essas características não se aplicam.

8.9. Exercícios Propostos

Exercício 1

Os iteradores são mesmo necessários para as listas? Explique.

Exercício 2

Vale a pena manter um cursor para uma lista implementada de forma contígua? Explique.

Exercício 3

Porquê que não existe uma método `add(Element)` na interface `ListADT`?

Exercício 4

Dada a versatilidade de uma lista porquê que não é sempre a escolhida em vez de uma `Stack` ou de uma `Queue`?

Exercício 5

Criar um método na `DoubleLinkedOrderedList` que inverta e devolva todos os elementos no tipo de dados abstrato que entender.

Exercício 6

Suponha que apenas tem uma `DoubleLinkedUnorderedList` e que pretende desenvolver um programa que precisa obrigatoriamente de um simples *array* para operar. Demonstre este simples cenário.

Exercício 7

A fórmula 1 é a prova de automobilismo mais conhecida e com mais prestígio a nível mundial. O campeonato mundial é constituído por diversas provas, equipas e corredores. Cada um dos corredores mediante a classificação em cada uma das provas irá obter pontos que irão formar a tabela classificativa e no final apurar o vencedor. Pretende-se que crie um programa que faça a simulação de uma tabela classificativa.