Aula 4
Tipos Abstratos de Informação e Coleções

Algoritmos e Estruturas de Dados

Tipos Abstratos

de Informação

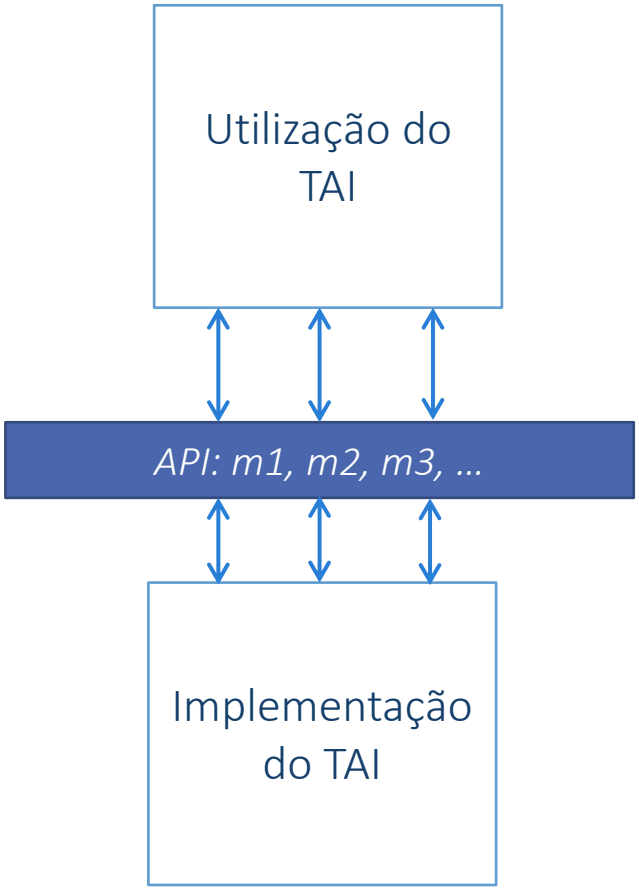
Tipos de dados

- Tipo de dados
  - *Def: **conjunto de valores** e **conjunto de operações** que se aplicam sobre esses valores*

Tipo	Valores	Operações	Exemplos de expressões
int	Inteiros entre -2^{31} e $+2^{31}-1$	+ (adição) - (subtração) * (multiplicação) / (divisão) % (resto da div inteira)	5 + 3 5 – 3 5 * 3 5 / 3 5 % 3
boolean	true ou false	&& (e) || (ou) ! (negação) ^ (xor – ou exclusivo)	true && false false || true !false true ^ true

Tipo Abstracto de Informação (TAI)

- Tipo Abstracto de Informação
  - ***Def:*** *tipo de dados cuja representação está "escondida" de quem os utiliza*
- Abstração de dados
  - ***Def:*** *Separação entre a utilização de um tipo de dados e sua implementação*



Tipo Abstracto de Informação (TAI)

- Assinatura de um tipo (***API*** – *Application Programming Interface*)
  - ***Def:*** *especificação de um tipo dados abstrato, indicando os seus métodos (entradas e saídas), com uma breve descrição informal do que cada um faz*

Contador	Tipo que representa um contador com estado que memoriza o número de vezes que foi incrementado	
`Contador`	`Contador(String id)`	*Cria um novo contador com o identificador id*
`void`	`incrementa()`	*Incrementa o contador em uma unidade*
`int`	`total()`	*Devolve o valor actual do contador*
`String`	`toString()`	*Devolve o contador na forma de uma cadeia de caracteres*

Tipos de Métodos TAI

Categoria	Descrição
Construtores	constroem novos elementos do tipo
Seletores	selecionam elementos constituintes do tipo
Modificadores	modificam componentes do tipo
Reconhecedores	reconhecem elementos do tipo, ou elementos específicos de um tipo
Comparadores	comparam elementos do tipo
Transformadores de saída	transformam elementos do tipo noutros tipos de dados
Transformadores de entrada	transformam outros tipos de dados num elemento do tipo

Exemplo API Tipo String em Java

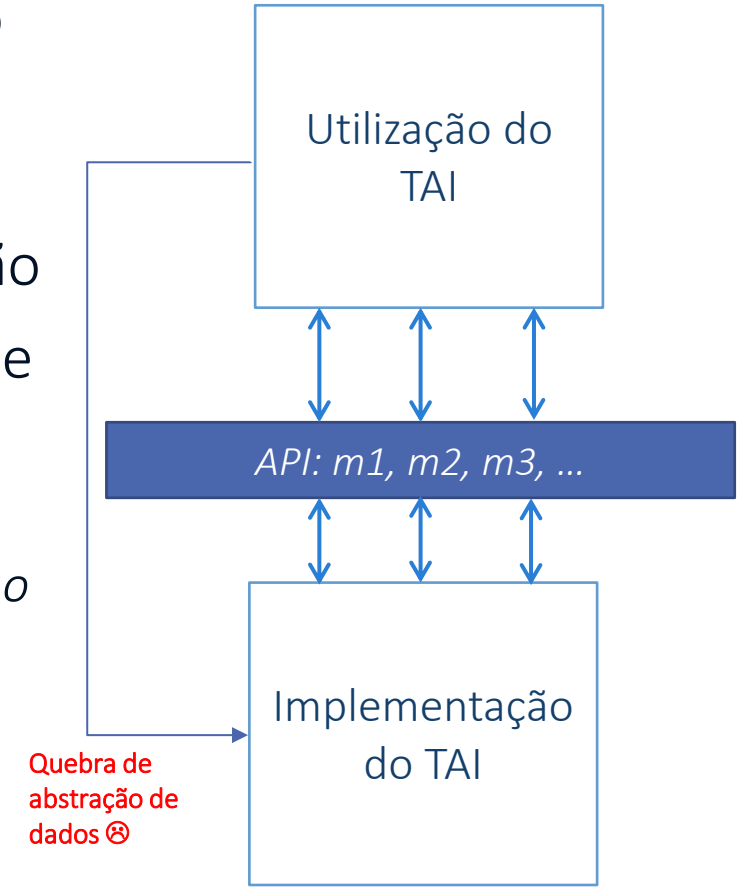
public class String	
String()	*create an empty string*
int length()	*length of the string*
int charAt(int i)	*i th character*
int indexOf(String p)	*first occurrence of* p *(-1 if none)*
int indexOf(String p, int i)	*first occurrence of* p *after* i *(-1 if none)*
String concat(String t)	*this string with* t *appended*
String substring(int i, int j)	*substring of this string (*i *th to* j*-1st chars)*
String[] split(String delim)	*strings between occurrences of* delim
int compareTo(String t)	*string comparison*
boolean equals(String t)	*is this string's value the same as* t*'s ?*
int hashCode()	*hash code*

Java String API (partial list of methods)

Quebra de abstração de dados

- **Evitar sempre quebras na abstração de dados**
  - Torna o código que utiliza o tipo dependente da sua implementação
  - Não é possível trocar facilmente de implementação do tipo

    Implica muitas vezes ter que mudar a implementação das funções que usam o Tipo de Dados



Abstração de Dados em Java

- A linguagem Java fornece vários mecanismos para garantir que as barreiras de abstração não são violadas

Modificadores de acesso em Java

public
private
protected
default

Permitem-nos separar aquilo que é publico (e pode ser usado) daquilo que é privado

Interfaces em Java

Mecanismo para especificação explicita de uma API

Separação explicita entre a API (Interface) e a sua implementação (Classes que implementam Interface)

O utilizador de uma Interface pode nunca chegar a saber que classe é que está por baixo da Interface

Coleções

Coleções

- Estruturas de dados usadas para representar
  - Grupos de objetos
  - Conjuntos de objetos
  - Listas ordenadas de objetos

Coleções em AED

- Em AED queremos estudar a fundo o funcionamento de vários tipos de coleções

  - Porquê?

Coleções em AED

- Em AED queremos estudar a fundo o funcionamento de vários tipos de coleções

  - Porquê?

  - Porque só com este conhecimento seremos capaz de

    As utilizar da forma mais apropriada para cada problema

    Implementar as nossas próprias coleções

Problema c/ Coleções Java

- ArrayList, LinkedList
  - Tentativa de implementar estruturas de dados muito generalistas que permitam trabalhar de várias formas

    Ex: ArrayList pode ser usado como:

    Lista

    Array

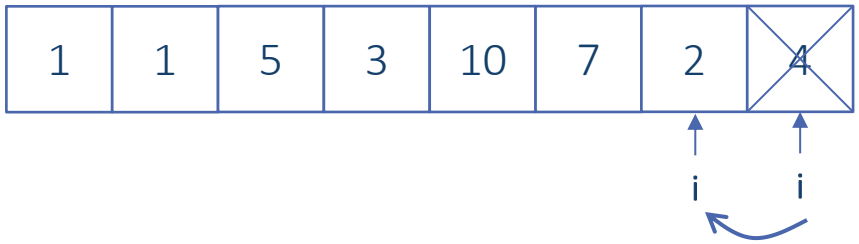
    Muitas ferramentas

    mas algumas são más!

    (em termos de eficiência)

Exemplo: remoção de elemento i ArrayList

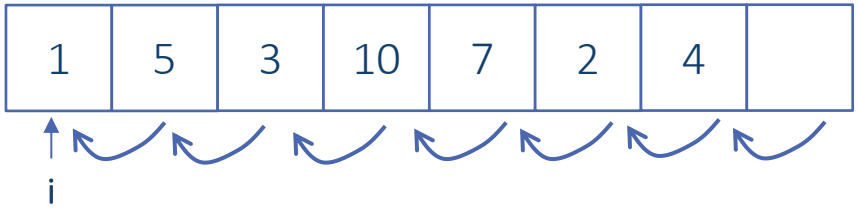
- Melhor caso, último elemento, i=7



- Operação muito eficiente, apenas precisamos de alterar um índice

Exemplo: remoção de elemento i ArrayList

- Pior caso, primeiro elemento, i=0



- Requer n operações no pior caso (n = tamanho do ArrayList)
  - Temos de fazer *"shift"* de todos os elementos do array

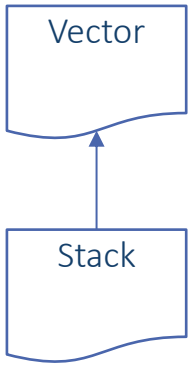
Problemas com ArrayList

- Arrays não são otimizados para
  - Remoções/adições no meio do Array

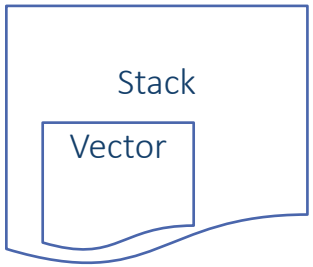
- má ideia disponibilizar métodos pensados para listas.

Problema c/Coleções Java

- Mau desenho (não é possível mudar)
  - Herança em vez de Composição



herança



composição

Java 1.3 bug report (June 27, 2001)
The iterator method on java.util.Stack iterates through a Stack from the bottom up. One would think that it should iterate as if it were popping off the top of the Stack.

Status (closed, will not fix)
It was an incorrect design decision to have Stack extend Vector ("is-a" rather than "has-a"). We sympathize with the submitter but cannot fix this because of compatibility.

Utilização de Coleções Java

- Usar as coleções do java quando
  - Eficiência não é relevante

    ou
  - A equipa de desenvolvimento conhece as coleções a fundo, sabe que coleção é a mais apropriada para cada situação, e tem a disciplina de não usar métodos ineficientes

  - Exemplo:

    Um bom eng. Informático, se estiver preocupado com eficiência, não deverá usar o método remove de um ArrayList

    Caso o tenham de fazer, devem pensar duas vezes se efetivamente um ArrayList é a estrutura de dados mais apropriada a usar

Eng. Informática e AED

- Para um Eng. Informático não basta saber "usar" as estruturas de dados do Java
  - Deve conhecê-las a fundo, e ser capaz de implementar as suas estruturas próprias, quando necessário.

    Por exemplo, quando existe uma necessidade de características particulares que nenhuma das estruturas implementadas nos dá

- A partir do 2.º projeto, vamos deixar de usar as coleções do Java, e aprender a criar as nossas coleções
  - **especializadas para maior eficiência**

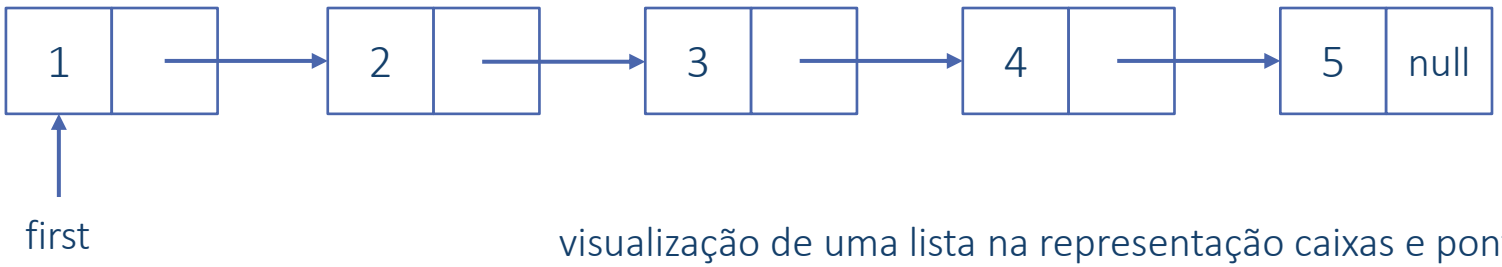
Listas Ligadas

Especificação e Implementação

Listas Ligadas

- Sequência de elementos ligados entre si
  - cada elemento tem uma ligação explícita para o seguinte
  - elementos podem ser acrescentados ou removidos facilmente no meio da lista

  - Ex: lista {1, 2, 3, 4, 5}

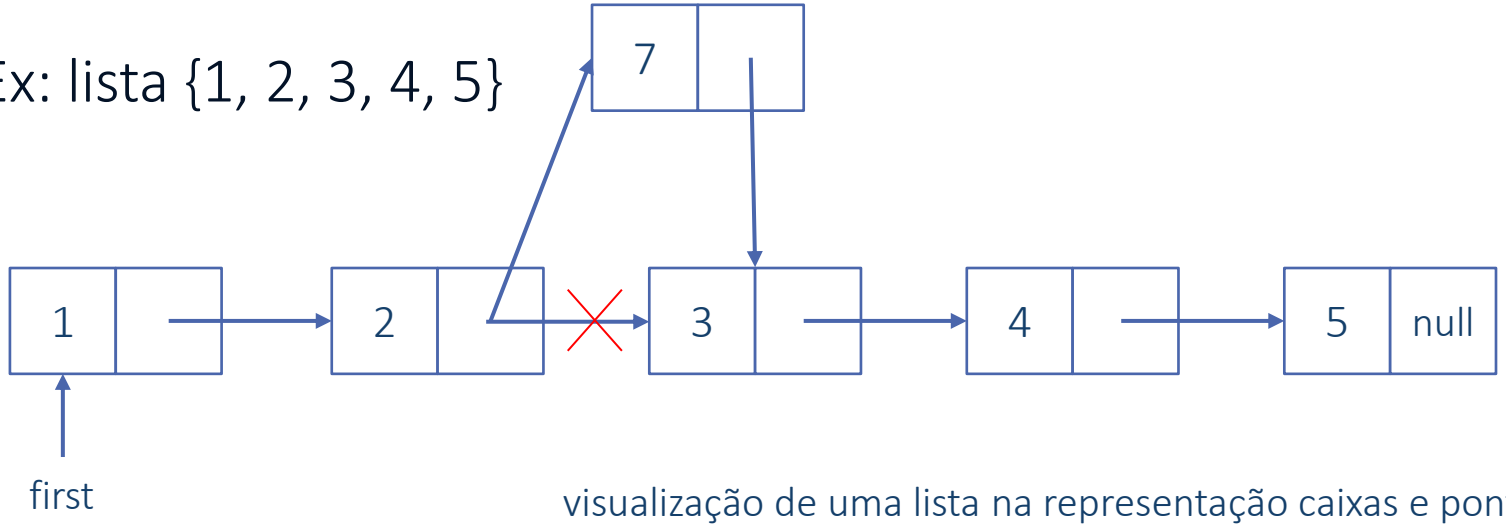


visualização de uma lista na representação caixas e ponteiros

Exemplo de inserção

- Sequência de elementos ligados entre si
  - cada elemento tem uma ligação explícita para o seguinte
  - elementos podem ser acrescentados ou removidos facilmente no meio da lista

- Ex: lista {1, 2, 3, 4, 5}



visualização de uma lista na representação caixas e ponteiros

Implementação de uma Lista

- Quando implementamos uma lista
  - Podemos escolher se a lista insere novos elementos

    no fim

    no início

  - Vamos ver uma implementação que insere elementos no início

    Implica que a ordem dos elementos vai ser contrária à ordem pela qual os inserimos

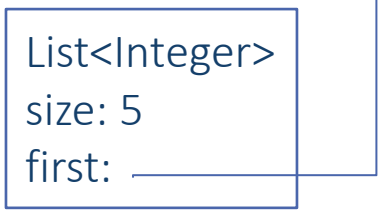
Nós de uma lista ligada


```
public class Node {
    private T item;
    private Node next;
}
```

- É possível implementar uma lista apenas como uma sequência de nós ligados

- No entanto, é vantajoso guardar informação adicional sobre a lista
  - Tamanho
  - Primeiro nó
  - Ultimo nó

Nós de uma lista ligada




```
private class Node {
    private T item;
    private Node next;
}
```

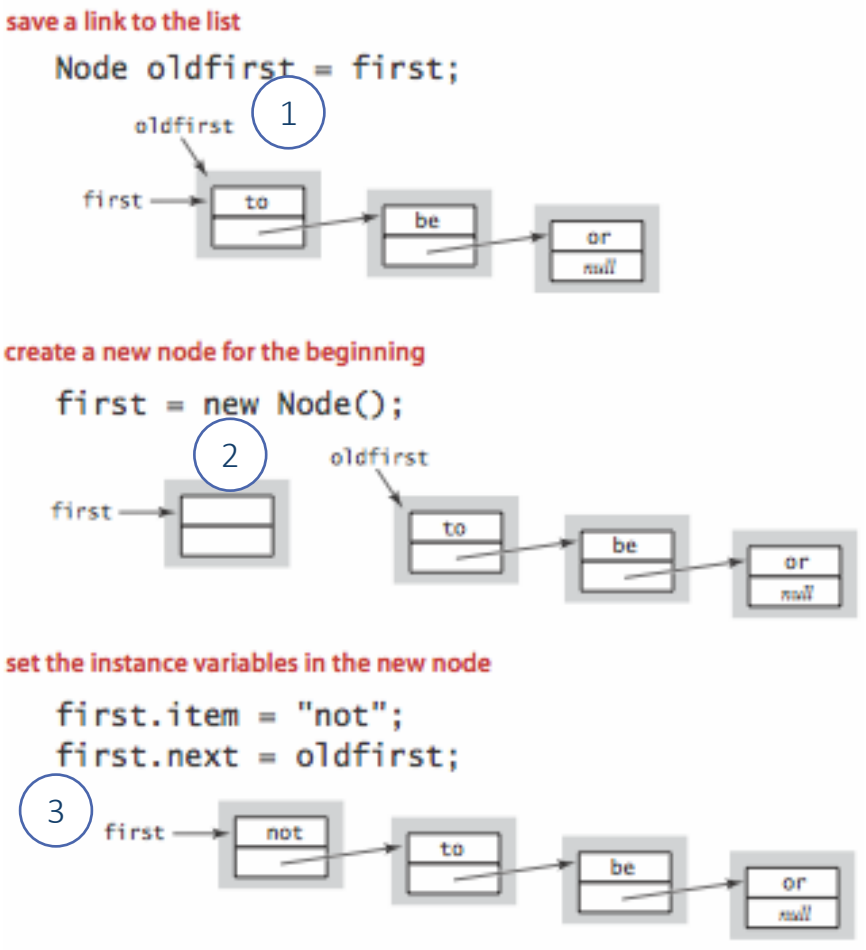
classe *Node* definida dentro da Lista

Lista – parâmetros e construtor

```
public class LinkedList<T> implements Iterable<T> {
   private class Node {
      private T item;
      private Node next;
   }

   private Node first;
   private int size;

   public LinkedList() {
      this.first = null;
      this.size = 0;
   }
```

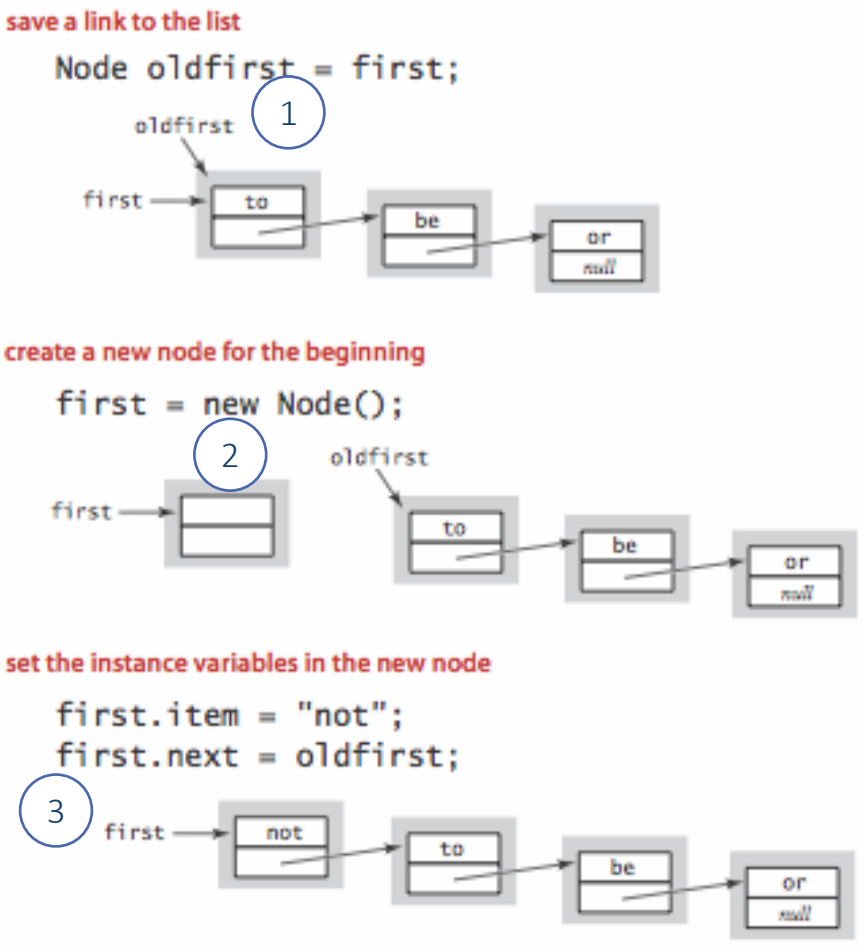
Lista – método *add*

1. guardar uma referência para o 1.º nó original
2. criar um novo nó
   1. colocar a lista a apontar para o novo nó
3. actualizar as variáveis do novo nó
4. Incrementar o tamanho



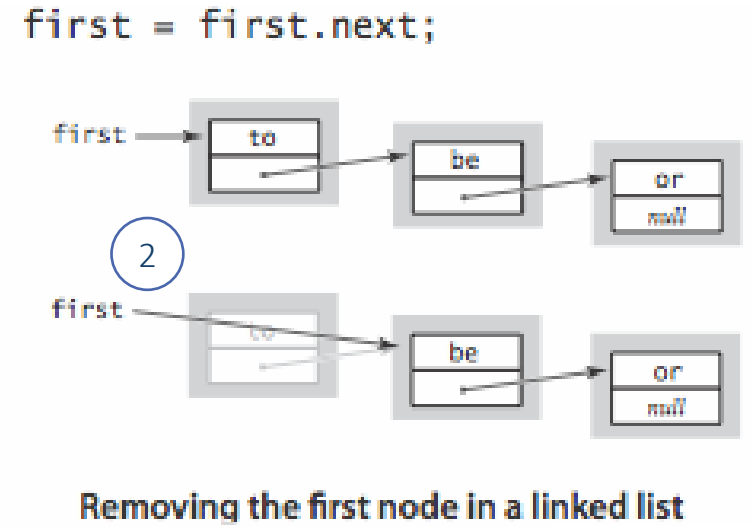
Lista – método *add*

```
public void add(T item)
{
    Node old = this.first;

    this.first = new Node();
    this.first.item = item;
    this.first.next = old;

    this.size++;
}
```



Lista – método *remove*

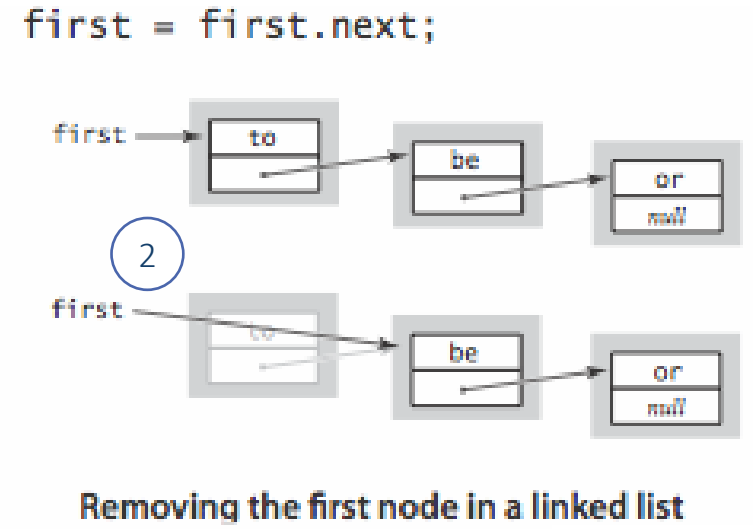
1. Guardar o item do 1.º nó
2. Actualizar a pilha para apontar para o 2.º nó
3. Diminuir tamanho
4. Retornar o item guardado



Removing the first node in a linked list

Lista – método *remove*

```
public T remove()
{
     if(this.first == null)
          return null;
     T result = this.first.item;
     this.first = this.first.next;
     this.size--;
     return result;
}
```



Removing the first node in a linked list

Lista – *Iterable* e *Iterator*

Iterator (Iterador):
objeto computacional usado para iterar sobre uma coleção ou sequência de elementos

Importante
Um iterador tem um estado (ou memória) que lhe permite recordar qual será o próximo elemento da coleção a ser iterado.

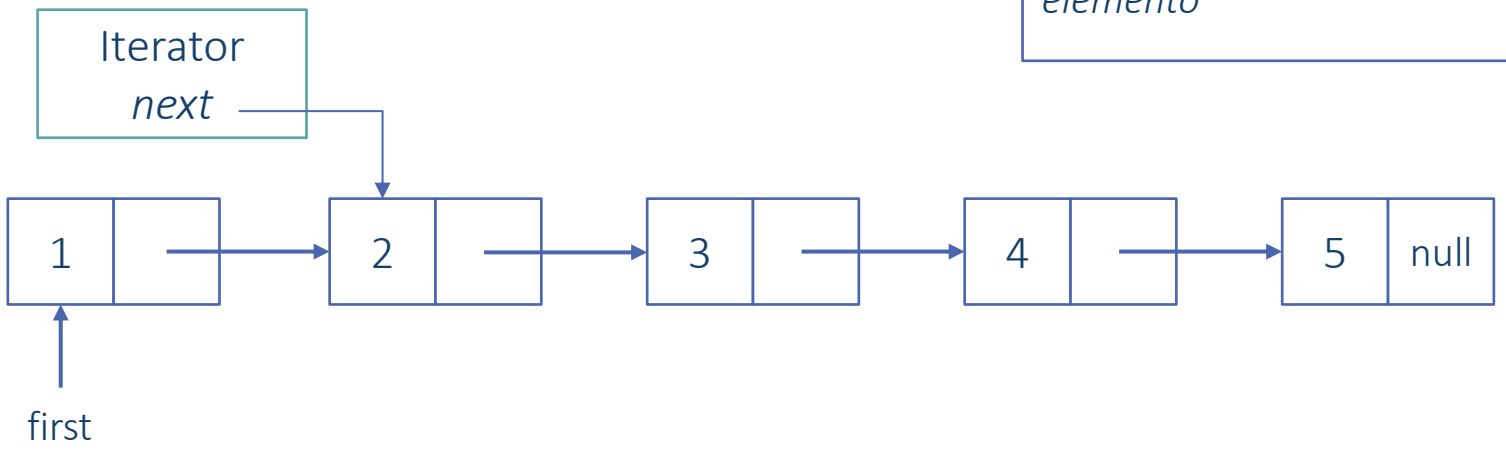
Eu posso pedir para fazer uma iteração, parar e só mais tarde voltar para pedir o próximo elemento

Lista – *Iterable* e *Iterator*

Iterator (Iterador):
objeto computacional usado para iterar sobre uma coleção ou sequência de elementos

Importante
Um iterador tem um estado (ou memória) que lhe permite recordar qual será o próximo elemento da coleção a ser iterado.

Eu posso pedir para fazer uma iteração, parar e só mais tarde voltar para pedir o próximo elemento



Lista – *Iterable* e *Iterator*

Iterator (Iterador):
objeto computacional usado para iterar sobre uma coleção ou sequência de elementos

Iterable (Iterável):
Coleção ou sequência de elementos que pode ser iterada. Para ser iterável, uma classe tem que ter um método que devolve um iterator. Esse iterator pode então ser usado para iterar.

Lista – *Iterable* e *Iterator*

- Iterador para listas
  - Começa no *first*
  - Vai avançado nó a nó (seguindo o campo *next*)
  - Até chegar ao fim da lista (*it == null*)

Lista – *Iterable* e *Iterator*

classe Iterator definida dentro da classe LinkedList

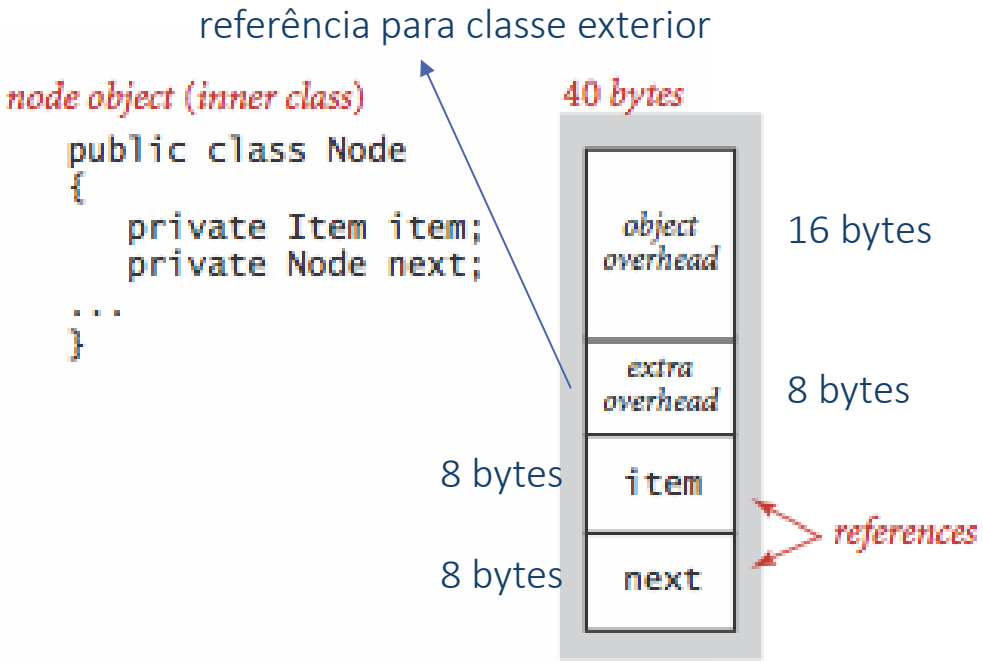
```
class LinkedListIterator implements Iterator<T>
{
    Node nextNode;
    LinkedListIterator()
    {
        nextNode = first;
    }
    boolean hasNext() {
        return nextNode != null;
    }
    T next()
    {
        T result = nextNode.item;
        nextNode = nextNode.next;
        return result;
    }
    void remove(){
        throw new UnsupportedOperationException("Iterator doesn't
                                                support removal");
    }
}
```

```
public Iterator<T> iterator()
{
    return new LinkedListIterator();
}
```

Este método faz com que a classe *LinkedList* possa ser considerada iterável

Lista – memória ocupada

- Complexidade espacial
  - 40 bytes por nó
  - 40 ***N*** bytes



Listas Ligadas vs Vetores

- Listas Ligadas
  - Estruturas de dados muito dinâmicas
  - Memória não contínua (versão tradicional)
  - Introdução ou remoção muito eficiente

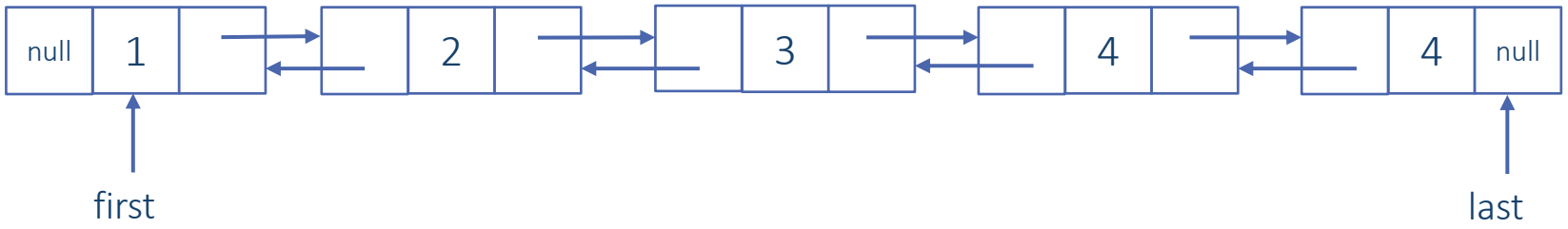
    início ou fim da lista – O(1)
  - Acesso a elementos no meio da lista não tão eficiente

- Vetores
  - Estruturas tradicionalmente mais estáticas com tamanho fixo
  - Zona de memória contínua
  - Acesso a qualquer elemento muito eficiente - O(1)
  - Introdução de elementos ou remoção ineficiente.

Lista duplamente ligada

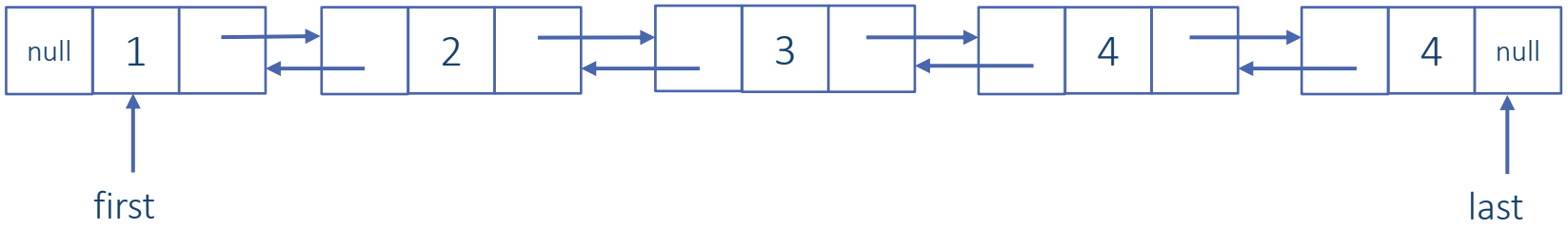
- Cada nó aponta para o próximo e para o anterior
- Lista tem um ponteiro para o primeiro e ultimo nó

  - Ex: lista {1, 2, 3, 4, 5}



Lista duplamente ligada

- Permite trabalhar de forma eficiente com
  - início
  - fim
- Devolver item da lista em metade do tempo
  - Se index < size/2, procurar do ínicio
  - procurar do fim c.c.



Lista duplamente ligada

- Desvantagens
  - Consome mais memória que uma lista ligada simples

    Lista simples – 40 bytes/nó

    Lista duplamente ligada – 48 bytes/nó