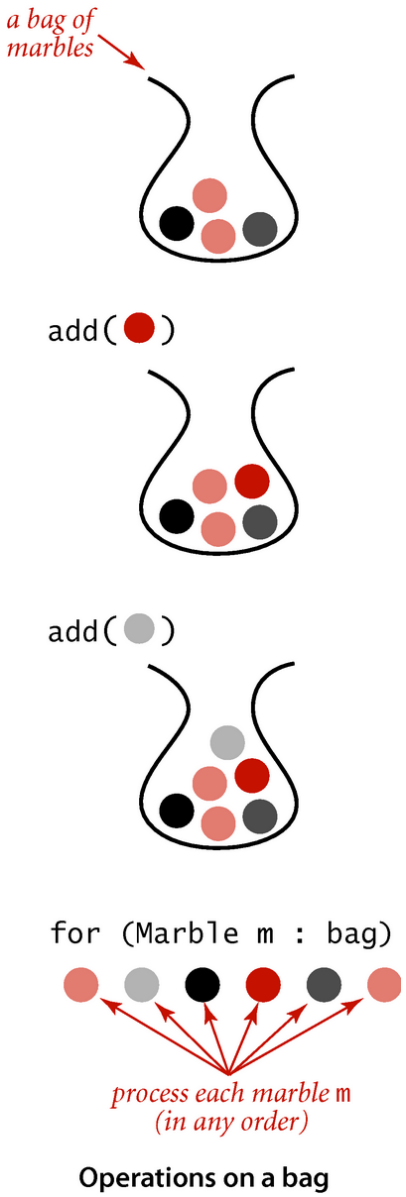
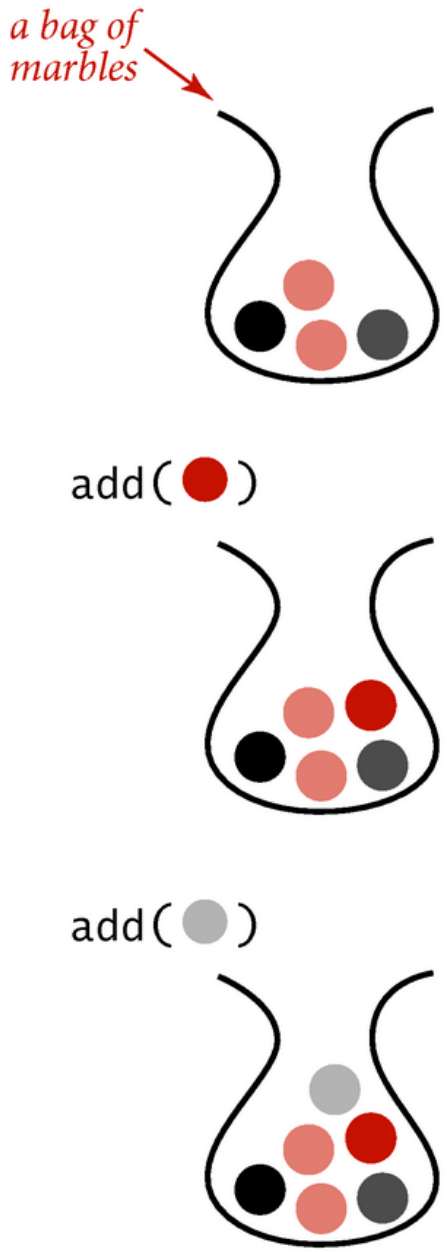
Sacos



Operations on a bag

Fonte: Saco (= bag) e sua API

Sacos



Que saco!

Um **saco** (=*bag*) é uma ADT que consiste de uma coleção de itens munida de duas operações:

- `add()` que insere um item na coleção, e
- `iterator()` que percorre os itens da coleção. A ordem em que o iterador percorre os itens não é especificada.

API de um saco de inteiros

public class	BagInteger	
	BagInteger()	cria um saco de inteiros vazio
void	add(Integer item)	coloca item neste saco
boolean	isEmpty()	este saco está vazio?
int	size()	número de itens neste saco
void	startIterator()	inicializa o iterador
boolean	hasNext()	há itens a serem iterados?
Integer	next()	próximo item

Cliente

```
public class Cliente {
   public static void main(String[] args){
       BagInteger bag = new BagInteger();
       for (int i=10; i < 20; i++) {
           bag.add(i);
       }
       StdOut.println(bag.size());
       bag.startIterator();
       while (bag.hasNext()) {
           StdOut.println(bag.next());
       }
   }
}
```

Class BagInteger: esqueleto

```
public class BagInteger {
   private Node first;
   private int n;
   private Node current;
   private class Node{...} //subclasse
   public BagInteger() {...} // construtor
   public void add(Integer item) {...}
   public int size() {...}
   public boolean isEmpty() {...}
   public void startIterator() {...}
   public boolean hasNext() {...}
   public Integer next() {...}
   public void remove() {...}
}
```

BagInteger: subclasse Node

```
private class Node{
   private Integer item;
   private Node next;
   public Node(Integer item, Node next) {
       this.item = item;
       this.next = next;
   }
}
```

BagInteger: construtor e add()

```
public BagInteger() { // construtor
   first = null;
}

public void add(Integer item) {
   Node oldfirst = first;
   first = new Node(item, oldfirst);
   // first.item = item;
   // first.next = oldfirst;
   n++;
}
```

```
public int size() {
   return n;
}
public boolean isEmpty() {
   return n == 0;
}
```

BagInteger: iterador

```
public void startIterator() {
   current = first;
}

public boolean hasNext() {
   return current != null;
}

public Integer next() {
   Integer item = current.item;
   current = current.next;
   return item;
}
```

API de um saco genérico

public class	Bag<Item>	
	Bag()	cria um saco de itens vazio
void	add(Item item)	coloca item neste saco
boolean	isEmpty()	este saco está vazio?
int	size()	número de itens neste saco
void	startIterator()	inicializa o iterador
boolean	hasNext()	há itens a serem iterados?
Item	next()	próximo item

Generics

Uma característica essencial de `ADTs` de coleções é permitir que sejam usadas para qualquer tipo de itens.

O mecanismo em `Java` conhecido como **genéricos** (=*generics*) permite essa capacidade.

A notação <`Item`> depois do nome da classe define o nome `Item` como um parâmetro de tipo, um espaço reservado para um tipo concreto ser usado pelo cliente.

Lemos `Bag`<`Item`> como *saco de itens* ou *bag de itens*.

Cliente

```
public class Cliente {
   public static void main(String[] args){
       Bag<String> bagS=new Bag<String>();
       bagS.add("Como "); bagS.add("é ");
       bagS.add("bom ");
       bagS.add("estudar ");
       bagS.add("MAC0323!");
       StdOut.println(bagS.size());
       bagS.startIterator();
       while (bagS.hasNext()) {
           StdOut.println(bagS.next());
       }
   }
}
```

Class Bag<Item>: esqueleto

```
public class Bag<Item> {
   private Node first;
   private int n;
   private Node current;
   private class Node{...} //subclasse
   public Bag() {...} // construtor
   public void add(Item item) {...}
   public int size() {...}
   public boolean isEmpty() {...}
   public void startIterator() {...}
   public boolean hasNext() {...}
   public Item next() {...}
   public void remove() {...}
}
```

Bag<Item>: subclasse Node

```
private class Node{
   private Integer item;
   private Node next;
   public Node(Integer item, Node next) {
       this.item = item;
       this.next = next;
   }
}
```

Bag<Item>: construtor e add()

```
public Bag() { // construtor
   first = null;
}

public void add(Item item) {
   Node oldfirst = first;
   first = new Node(item, oldfirst);
   // first.item = item;
   // first.next = oldfirst;
   n++;
}
```

Bag<Item>: size() e isEmpty()

```
public int size() {
   return n;
}

public boolean isEmpty() {
   return n == 0;
}
```

Bag<Item>: iterador

```
public void startIterator() {
   current = first;
}

public boolean hasNext() {
   return current != null;
}

public Item next() {
   Item item = current.item;
   current = current.next;
   return item;
}
```

Iteradores

`API` permite apenas um iterador:

Iteradores

queremos vários:

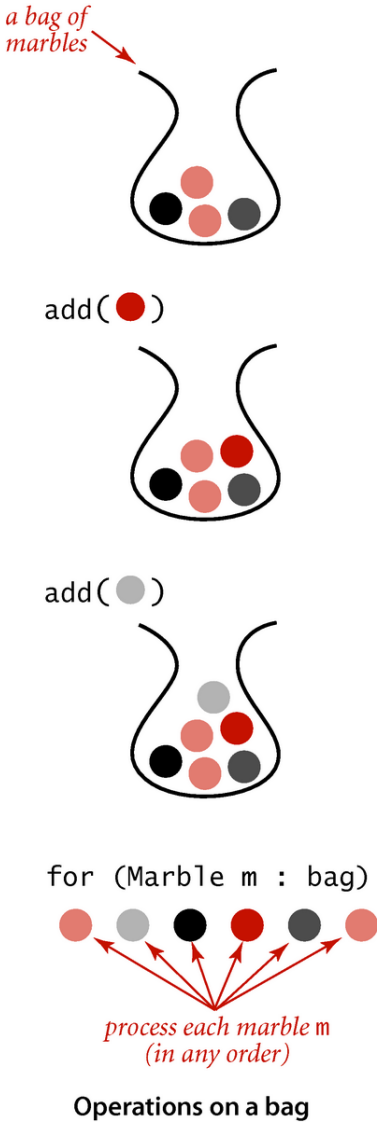
API: saco genérico iterável

public class	Bag<Item>	implements iterable<Item>
	Bag()	cria um saco de itens vazio
void	add(Item item)	coloca item neste saco
boolean	isEmpty()	este saco está vazio?
int	size()	número de itens neste saco
iterator<Item>	iterator()	iterador de itens

Cliente

```
public class Cliente {
   public static void main(String[] args){
       Bag<String> bagS=new Bag<String>();
       bagS.add("Como "); bagS.add("é ");
       bagS.add("bom ");
       bagS.add("estudar ");
       bagS.add("MAC0323!");
       StdOut.println(bagS.size());
       Iterator<String> it =
                  bagS.iterator();
       while (it.hasNext()) {
           StdOut.println(it.next());
       }
   }
```

foreach



Operations on a bag

Frequentemente o cliente precisa apenas processar cada item de uma coleção iterável de alguma maneira. Para isso podemos iterar sobre os items da coleção com um comando do tipo foreach.

```
Bag<String> bagS =
        new Bag<String>();
[...]
for (String s:  bagS)
   StdOut.println(s);
```

Cliente de Luxe

```
public class Cliente {
   public static void main(String[] args){
       Bag<String> bagS=new Bag<String>();
       bagS.add("Como "); bagS.add("é ");
       bagS.add("bom ");
       bagS.add("estudar ");
       bagS.add("MAC0323!");
       StdOut.println(bagS.size());
       for (String s:  bagS) {
           StdOut.println(s);
       }
   }
}
```

Receita para construir uma classe iterável

Leia Bags, Queues, and Stacks (SW)
ou Saco (= bag) e sua API (PF).

Passo 0: incluir

```
import java.util.Iterator;
```

para que possamos nos referir a interface
`java.util.Iterator`:

Receita para construir uma classe iterável

Passo 1: adicionar no final da declaração da classe `implements Iterable<Item>`.
Isso indica que o objeto será iterável e nos comprometemos a especificar o método `iterator()`, como especificado na interface `java.lang.Iterable`

```
public interface Iterable<Item> {
   public Iterator<Item> iterator();
}
```

Por exemplo:

```
public class Bag<Item> implements Iterable<Item>{
   [...]
}
```

Receita para construir uma classe iterável

Passo 2: implementar um método `iterator()` como prometido. Esse método retorna um objeto da classe que implementa a interface `Iterator`

```
public interface Iterator<Item> {
   boolean hasNext();
   Item next();
   void remove();
}
```

Por exemplo:

```
public Iterator<Item> iterator() {
   return new BagIterator();
}
```

Receita para construir uma classe iterável

Passo 3: implemente a subclasse que implementa a interface `Iterator` incluindo os método `hasNext()`, `next()` e `remove()`

Usamos sempre o método vazio para o opcional método `remove()` pois intercalar iteração com uma operação que modifica a estruturas de dados é melhor ser evitada.

Receita para construir uma classe iterável

```
private class
BagIterator implements Iterator<Item> {
   private Node current = first;
   public boolean hasNext() {
       return current != null;
   }
   public Item next() {
       Item item = current.item;
       current = current.next;
       return item;
   }
   public void remove() {
       throw new UnsupportedOperationException();
   }
```

Observações

Fonte: filmeseriale.info

Ao longo do semestre usaremos Bags frequentemente.

Em particular, usaremos Bags em uma das nossas implementações de *grafos* e *digrafos*

Listas encadeadas em Java

SW 1.3

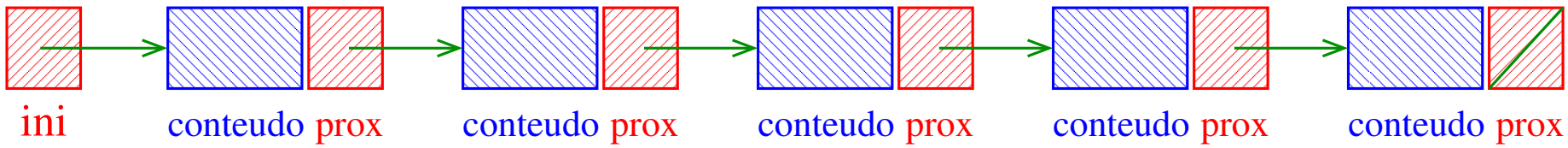
https://algs4.cs.princeton.edu/13stacks/

Linked lists, Victor S.Adamchik, CMU, 2009

Listas encadeadas

Uma **lista encadeada** (= *linked list* = lista ligada) é uma sequência de células; cada célula contém um objeto de algum tipo e o endereço da célula seguinte.

Ilustração de uma lista encadeada ("sem cabeça")



Estrutura para listas encadeadas em Java

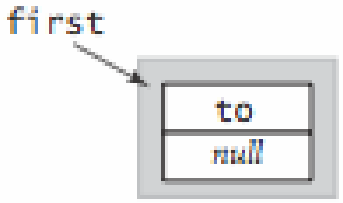
É conveniente tratar as células como um novo tipo-de-dados e atribuir um nome a esse novo tipo:

```
private class Node{
   Item item;
   Node next;
}
first = null;
```

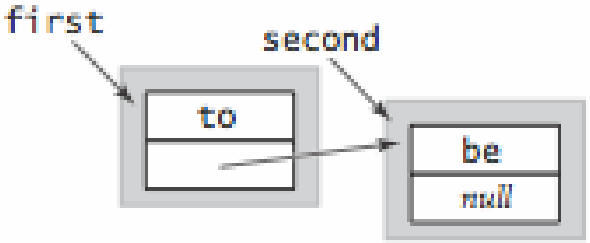
ini

Construir uma lista ligada

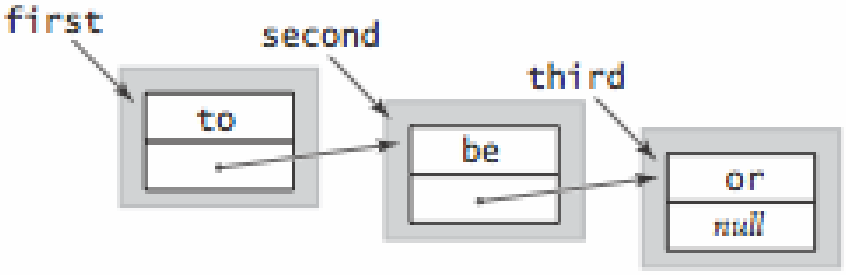
```
Node first  = new Node();
first.item  = "to";
```




```
Node second = new Node();
second.item = "be";
first.next  = second;
```



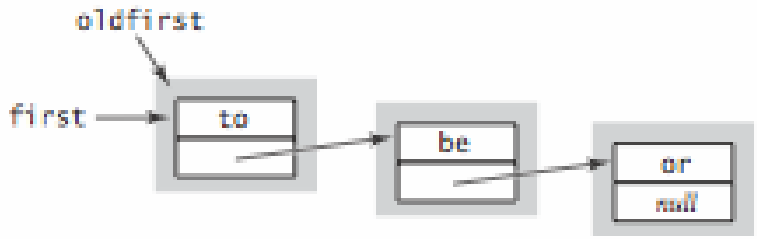

```
Node third  = new Node();
third.item  = "or";
second.next = third;
```



Inserir no início

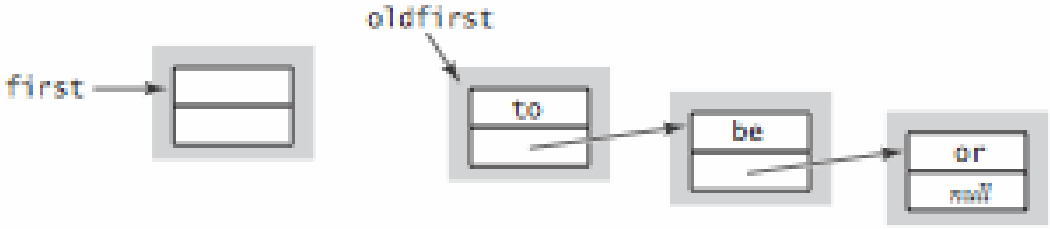
save a link to the list

```
Node oldfirst = first;
```



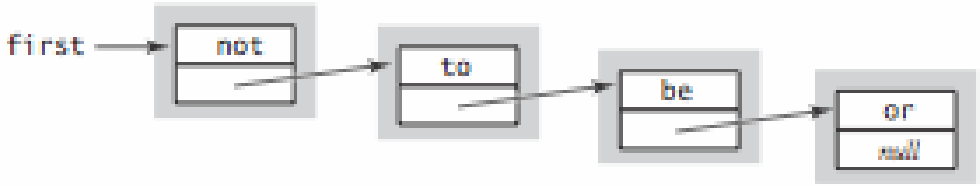
create a new node for the beginning

```
first = new Node();
```



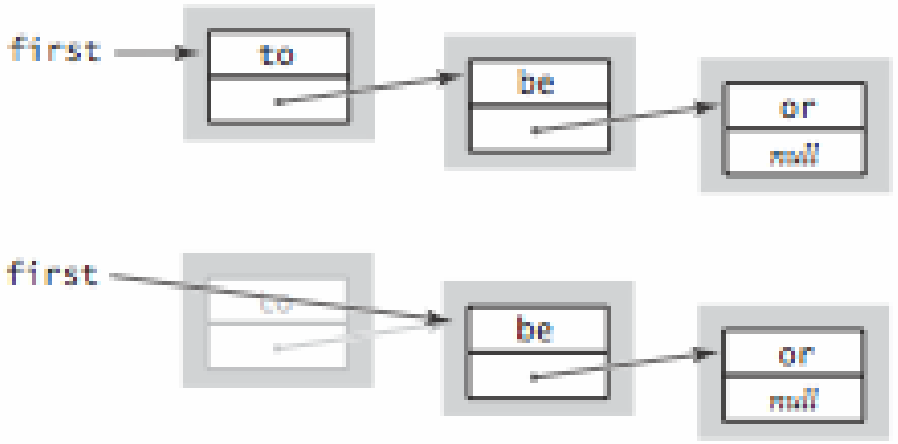
set the instance variables in the new node

```
first.item = "not";
first.next = oldfirst;
```



Remover do início

```
first = first.next;
```

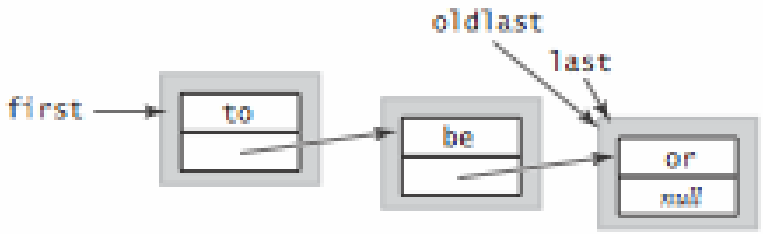


Removing the first node in a linked list

Inserir no final

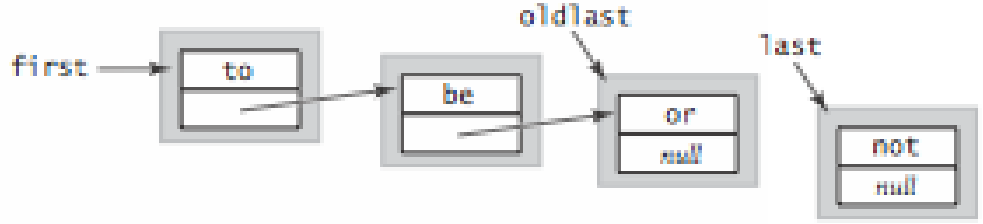
save a link to the last node

```
Node oldlast = last;
```



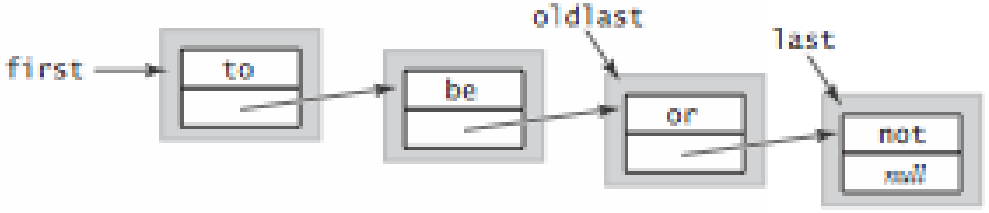
create a new node for the end

```
Node last = new Node();
last.item = "not";
```



link the new node to the end of the list

```
oldlast.next = last;
```



Percorrer

O seguinte trecho de código percorre uma lista ligada.

```
for (Node x = first; x != null; x = x.next)
{
   // processe x.item
}
```