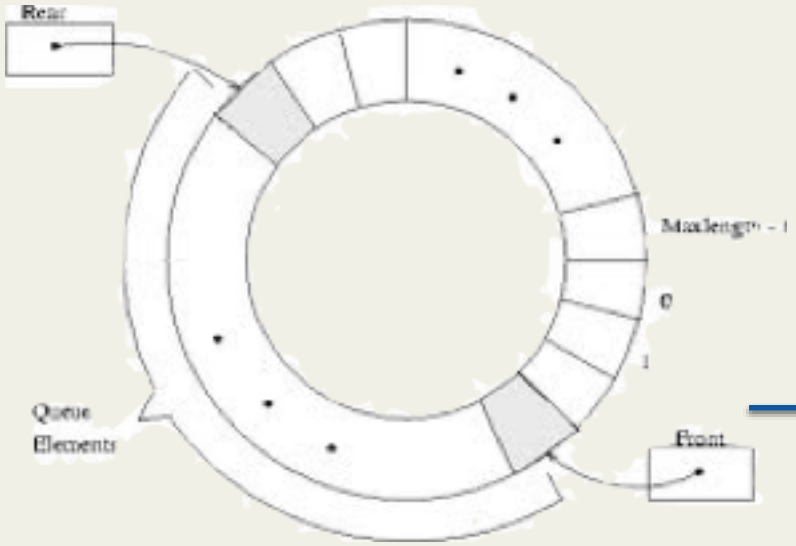
Ciências ULisboa

Faculdade de Ciências da Universidade de Lisboa



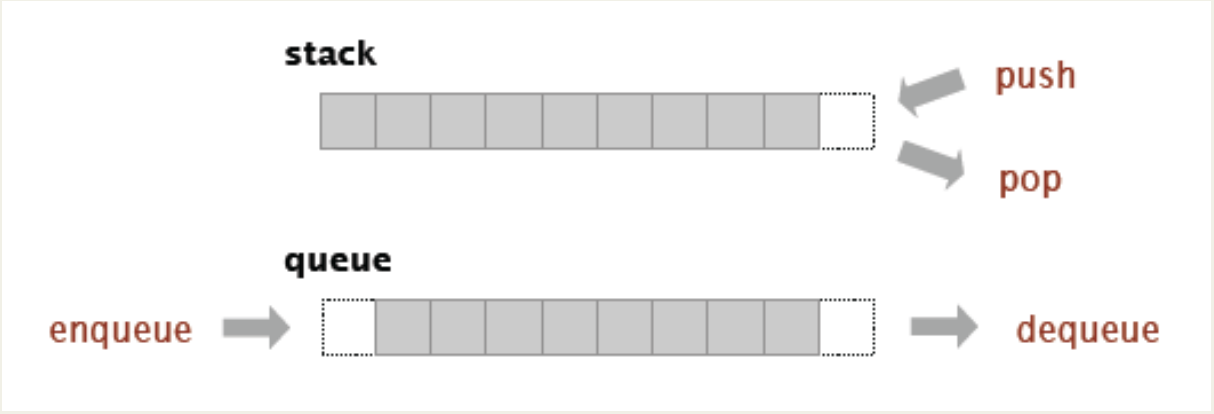

# Filas e Listas

- *(Stack)*
- *Queue*
- *Deque*
- *List*
- Iteradores

# Queue

Uma fila de elementos com filosofia **FIFO** (*first in first out*)

- **make**: constrói fila sem nenhum elemento
- **enqueue**: insere no fim da fila um elemento dado
- **dequeue**: retira o elemento que está na frente da fila
- **front**: indica o elemento na frente da fila
- **isEmpty**: indica se a fila está vazia



# Queue: Especificação

```
specification Queue[Element]
  sorts
    Queue[Element]
  constructors
    make: --> Queue[Element];
    enqueue: Queue[Element] Element --> Queue[Element];
  observers
    front: Queue[Element] -->? Element;
    dequeue:  Queue[Element] -->? Queue[Element];
    isEmpty: Queue[Element];
  domains
    Q: Queue[Element];
    front(Q)     if not isEmpty(Q);
    dequeue(Q)   if not isEmpty(Q);
  axioms
    Q: Queue[Element]; E: Element;
    front(enqueue(Q, E)) = E when isEmpty(Q) else front(Q);
    dequeue(enqueue(Q, E)) = Q when isEmpty(Q)
        else enqueue(dequeue(Q), E);
    isEmpty(make());
    not isEmpty(enqueue (Q, E));
end specification
```

# Queue: Implementação (API)

```
public class MyQueue<E>
        MyQueue<E>()      Create an empty queue.
   void enqueue(E e)      Insert an element at the rear of the queue.
   void dequeue()         Remove the front element from the queue. Requires
                          the queue is not empty.
      E front()           Inspect the element at the front of the queue.
                          Requires the queue is not empty.
boolean isEmpty()         Return whether the queue is empty.
```

# Queue: Refinamento

```
refinement<E>
    Element is E

    Queue[Element] is MyQueue<E> {
          make: --> Queue[Element]
                      is MyQueue ();
          enqueue: Queue[Element] e:Element --> Queue[Element]
                      is void enqueue(E e);
          front: Queue[Element] -->? Element
                      is E front();
          dequeue: Queue[Element] -->? Queue[Element]
                      is void dequeue();
          isEmpty: Queue[Element]
                      is boolean isEmpty();
    }
end refinement
```

## Implementação de ADTs: Interfaces

- A implementação de um ADT em Java envolve em geral ainda um outro passo

  a definição de um **interface** Java

- Este interface tem **toda a informação da API sobre métodos** (assinaturas, javadoc, contratos). De fora ficam
  - os construtores
  - e o nome da classe

- A classe que implementa o ADT passa a declarar implementar este interface

## Interface Java

- Um **interface** Java contém **declarações de assinaturas** de métodos públicos e declarações de constantes e define um tipo abstracto
  - ao contrário das classes não podem ser instanciados
- A partir da versão 8, os interfaces Java tornaram-se mais ricos e passaram a poder incluir mais algumas coisas (métodos de classe, métodos *default*, tipos aninhados)
- Uma classe declara implementar um ou mais interfaces fazendo uso da *keyword* `implements`
  - isto implica que todos os métodos declarados nesses interfaces estão necessariamente implementados na classe

# Queue: Implementação (interface)

```
interface Queue<E> {
   /** Insert an element at the rear of this queue.
   * @param e The object to insert.
   */
   public void enqueue(E e);
   /** Remove the head of the queue.
   * @requires !isEmpty()
   */
   public void dequeue();
   /**
   * @return e The head of this queue.
   * @requires !isEmpty()
   */
   public E front();
   /**
   * @return If this queue is empty.
   */
   public boolean isEmpty();
}
```

# Queue: Implementação (interface)

```
/**
* Interface for a mutable queue: a collection of objects that are inserted
* and removed according to the first-in first-out principle
*/
interface Queue<E> {
  /**
  * Insert an element at the tail of the Queue.
  * @param e The object to insert.
   */
  public void enqueue(E e);
  ...
}
```

```
public class MyQueue<E> implements Queue<E>{
    ...
  /**
  * (non-Javadoc)
  * @see Queue#enqueue()
   */
  public void enqueue(E e){
    if (...)
        ...
    }
}
```

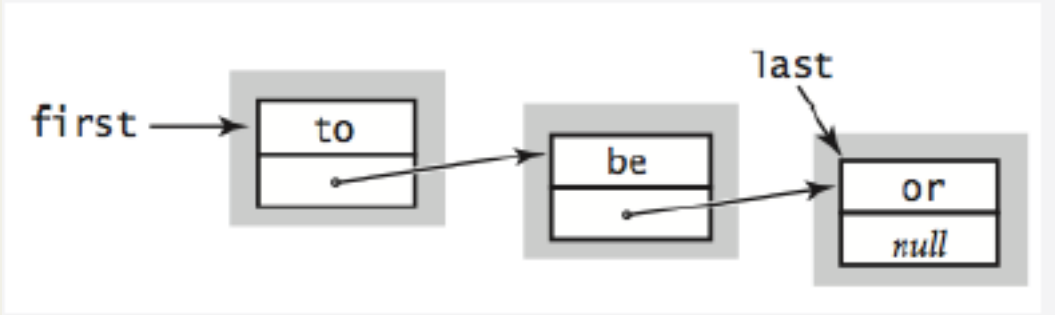
# Queue: Implementação com lista ligada

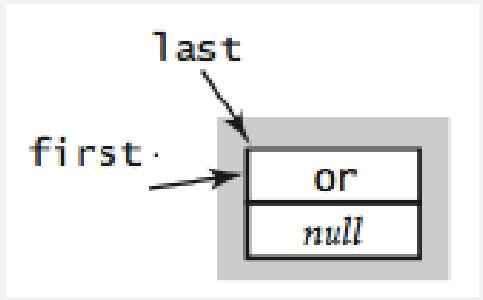
# Queue: Implementação com lista ligada

Escolhendo como estrutura de dados a **lista ligada** a ideia é:

- manter em ***first*** uma referência para o primeiro nó da lista ligada
- manter em ***last*** uma referência para o último nó da lista ligada
- remover da frente da lista
- inserir no fim da lista

```
Node<E> first;
Node<E> last;
```

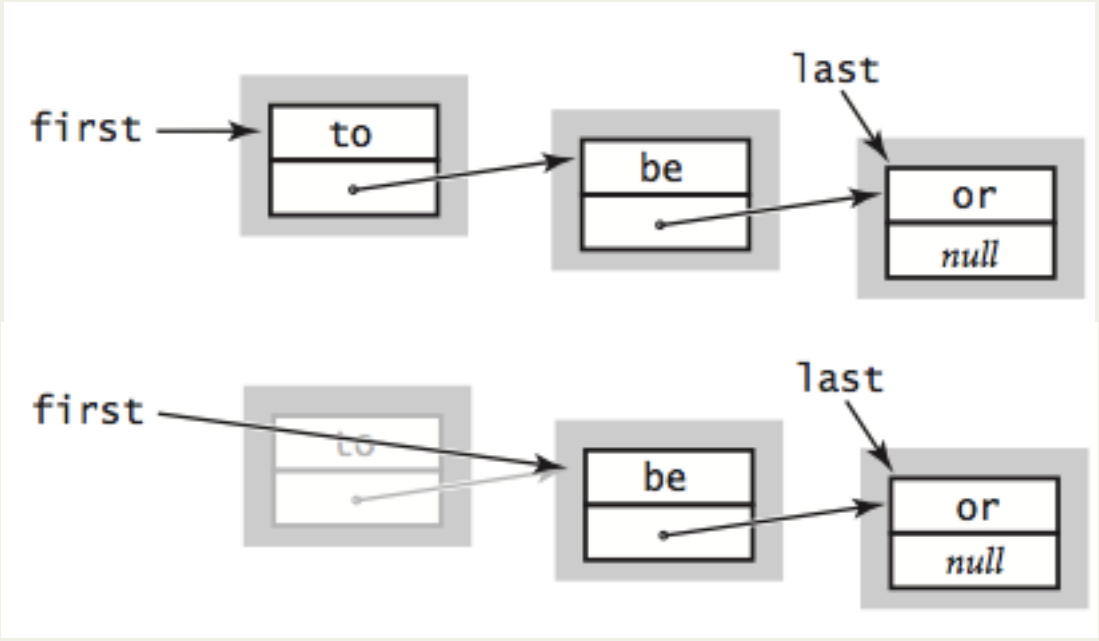


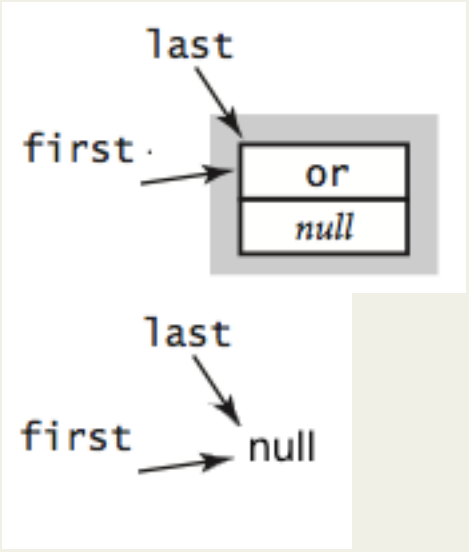



```
public class LLQueue<E> implements Queue<E>{

      //inner class Node
       private class Node<E> {
       private E item;
       private Node<E> next;

       private Node(E item, Node<E> next){
         this.item = item; this.next = next;
      }
    }

    Node<E> first;
    Node<E> last;

      ...
}
```

# Queue: Implementação com lista ligada

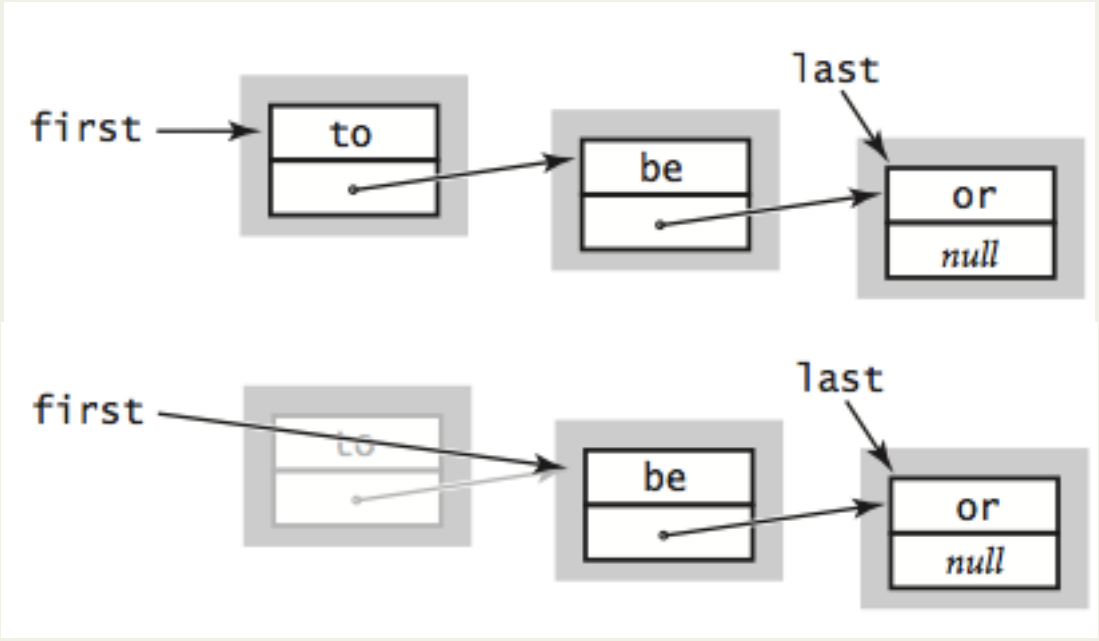
```
void dequeue()
```

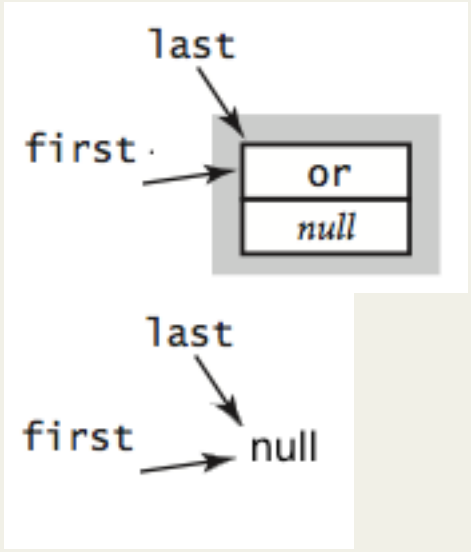


# Queue: Implementação com lista ligada

`void dequeue()`

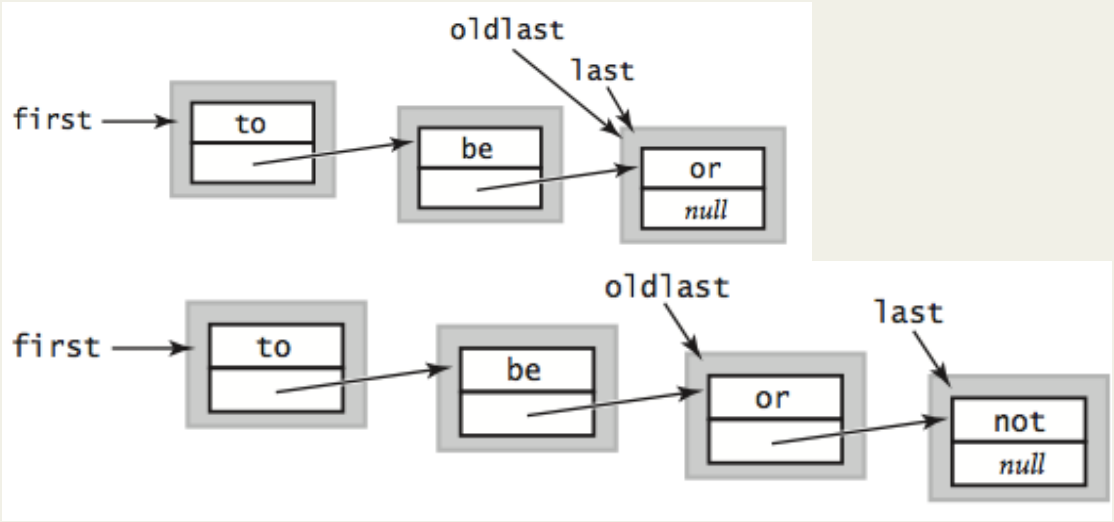


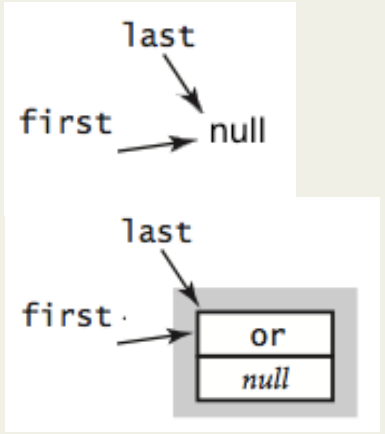




```
public void dequeue() {
   first = first.next;
    if (first == null)
        last = null;
}
```

# Queue: Implementação com lista ligada

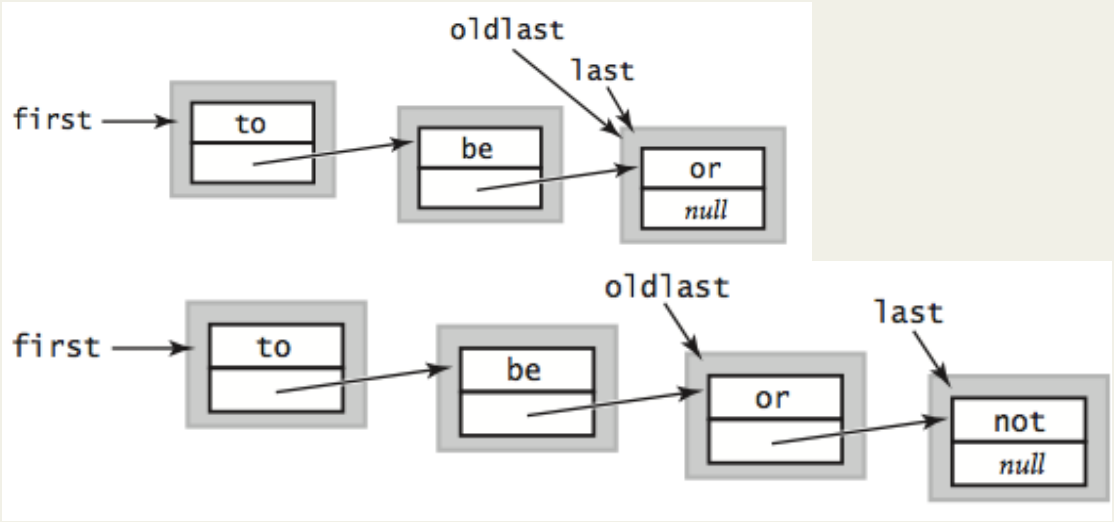
```
void enqueue(E element)
```

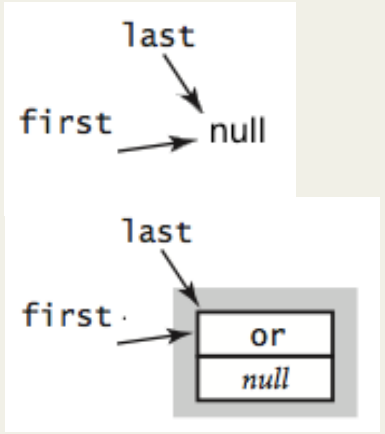


# Queue: Implementação com lista ligada

```
void enqueue(E element)
```



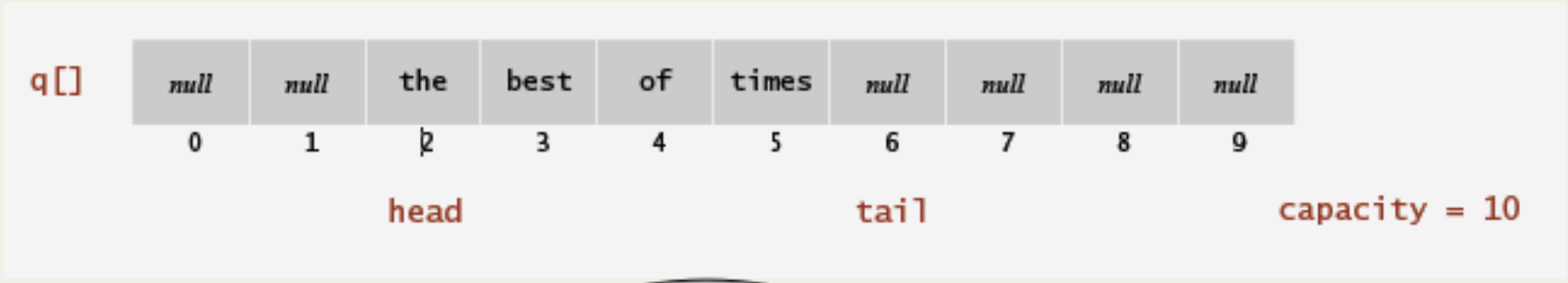

```
public void enqueue(E element) {
    Node<E> oldLast = last;
    last = new Node<E>(element,null);
    if (first == null)
        first = last;
    else
        oldLast.next = last;
 }
```

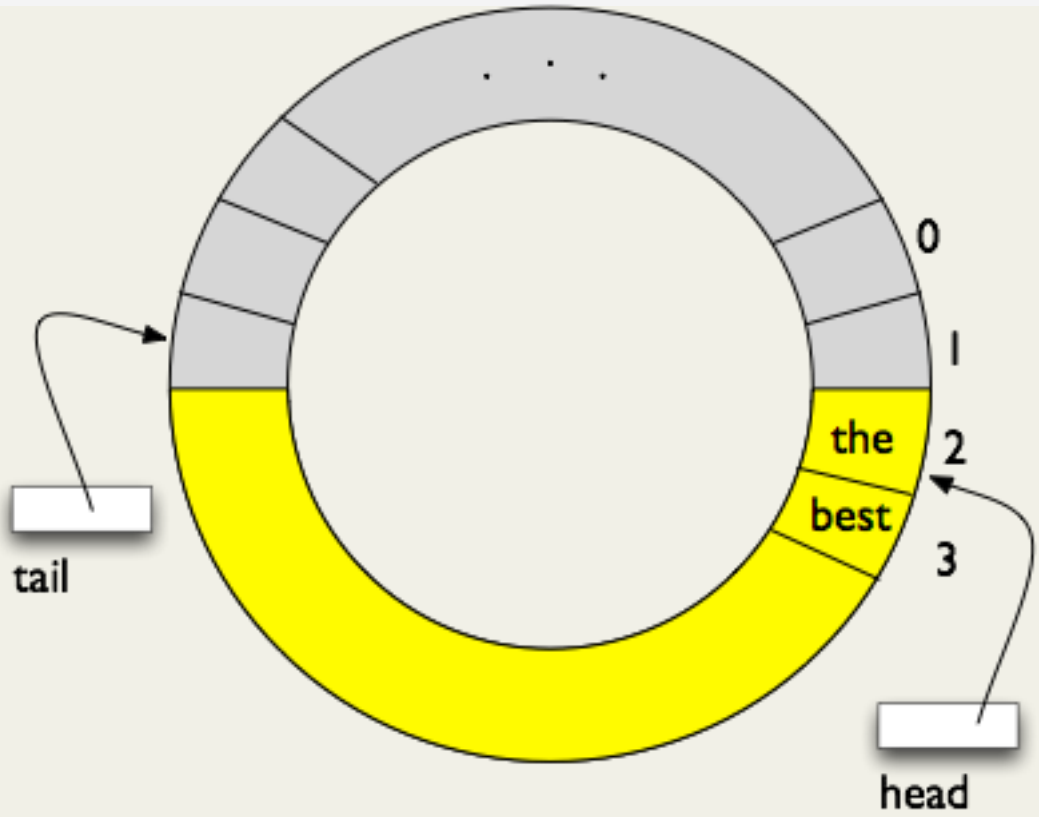
# Queue: Implementação com vector

# Queue: Implementação com vector circular

- Escolhendo como estrutura de dados um **vector circular** a ideia é ter:
  - vector `q[]` que guarda os elementos
  - `head` que guarda o índice do primeiro elemento
  - `tail` que guarda o índice da cauda (a seguir ao último)

# Queue: Implementação com vector circular



# Queue: Implementação com vector circular

- `void enqueue(E element)`
  - adiciona elemento em `q[tail]`
  - actualiza `tail`
- `void dequeue()`
  - remove elemento de `q[head]`
    - actualiza `head`

A actualização de `head` e `tail` é feita com operações módulo a capacidade do vector (ex., `(tail+1)%10` )

**O que acontece se o vector está cheio?**

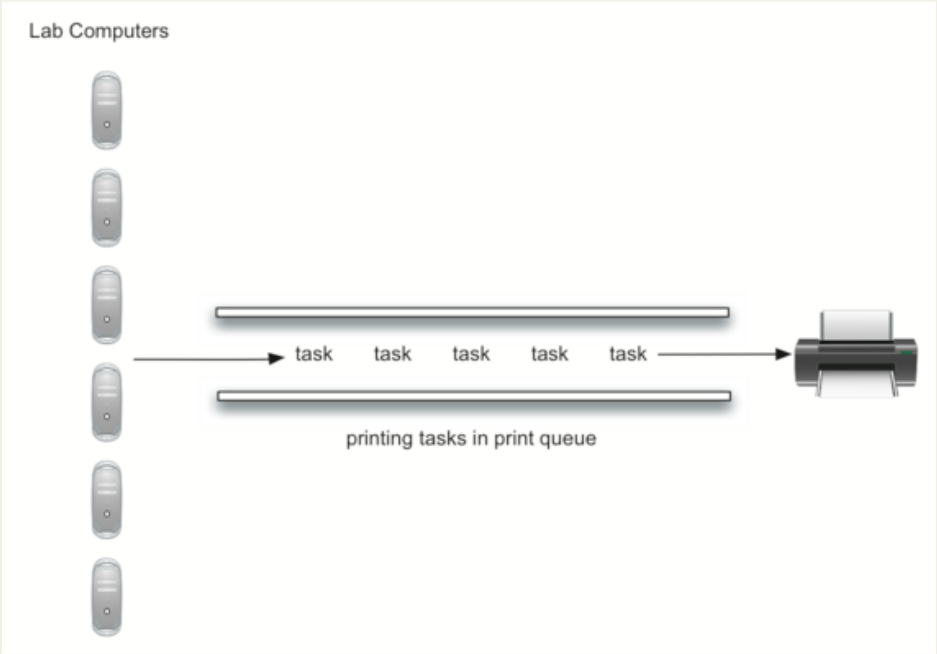
# Queue: Implementação com vector circular

- De forma a não ter restrições relativamente à inserção de elementos é preciso fazer crescer o vector quando se vai inserir um elemento e o vector já está cheio (`head==tail`) tal como em `ArrayStack`
- Podemos usar um **vector ajustável** que
  - cresce para o dobro quando cheio (*enqueue*)
  - reduz para metade quando só tem 1/4 ocupado (*dequeue*)

| Method | Time |
|---|---|
| front | O(1) |
| dequeue | O(1) amortizado |
| enqueue | O(1) amortizado |
| isEmpty | O(1) |

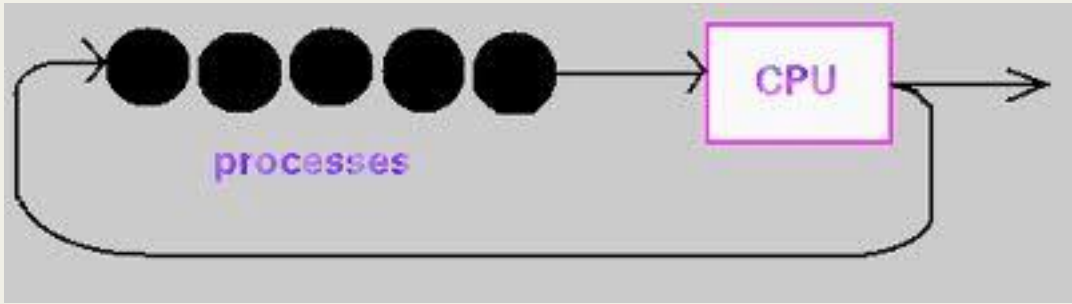
o tempo de execução de uma sequência de *n* operações de *enqueue*, *dequeue* sobre *fila* que começa vazia é *O(n)*

# Aplicações



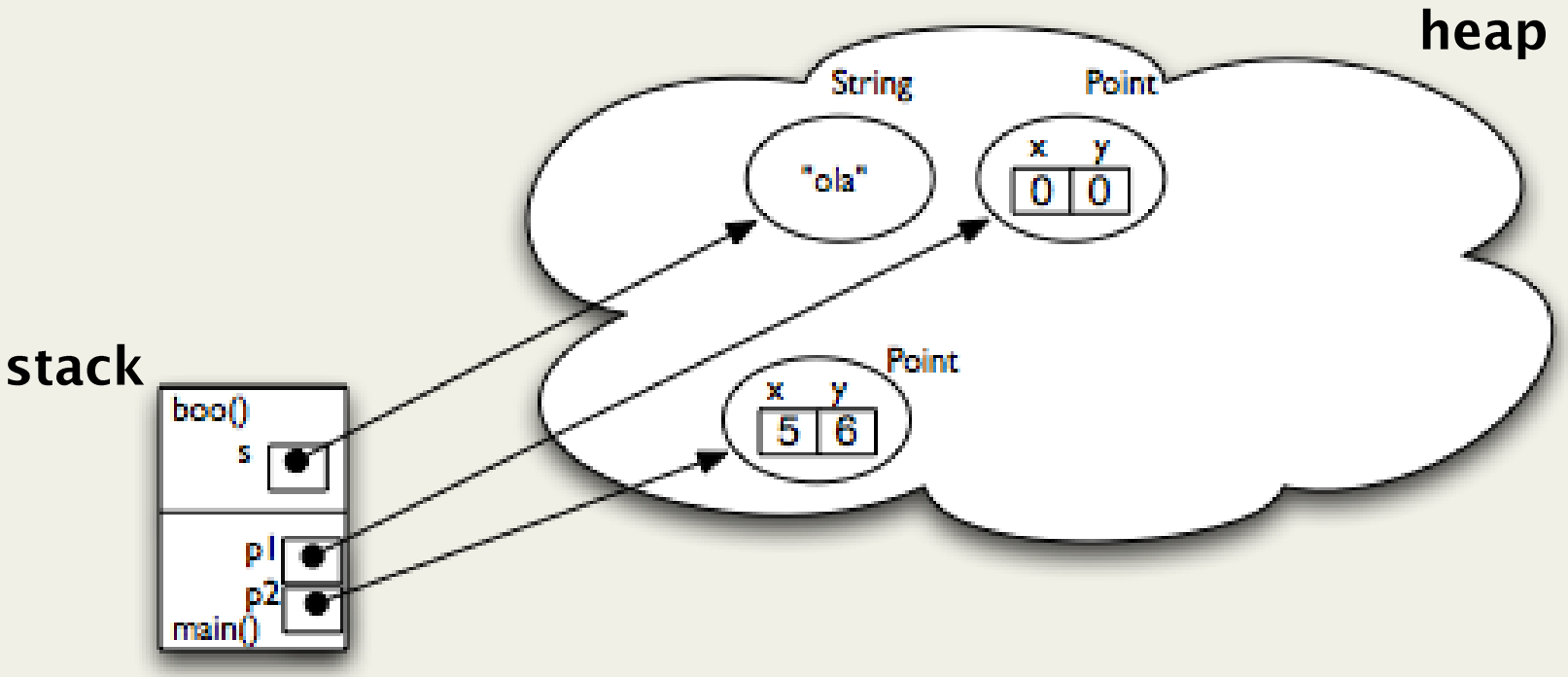

## Servidor de Impressões

## Round robin scheduler



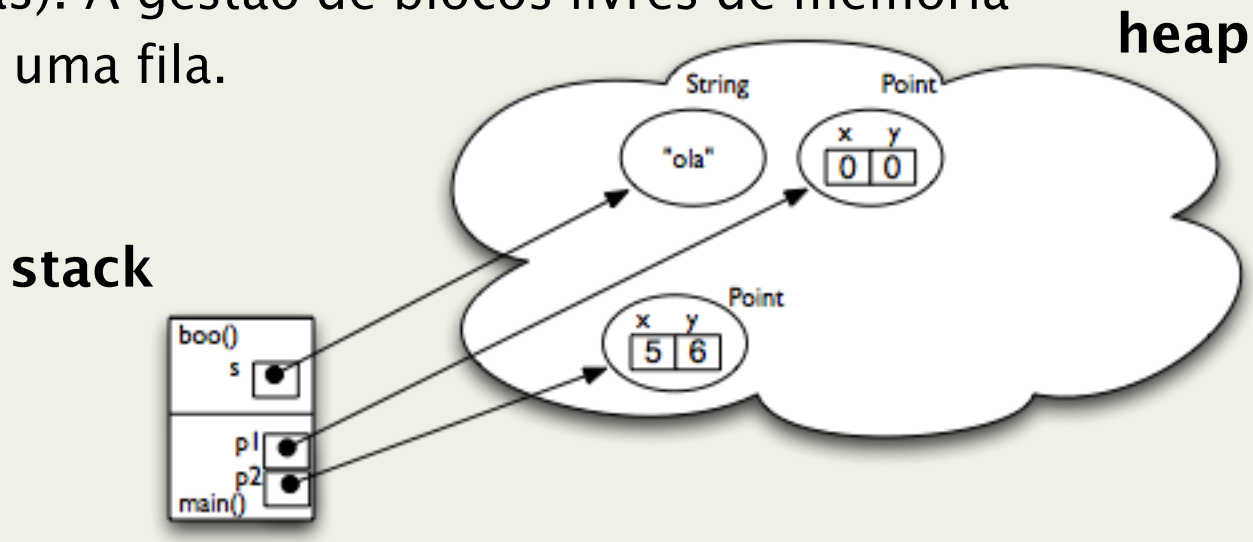
# Alocação de memória em Java

- Anteriormente vimos como a JVM trata de arranjar espaço para as variáveis locais de um método que foi invocada e as coloca na ***Java stack***



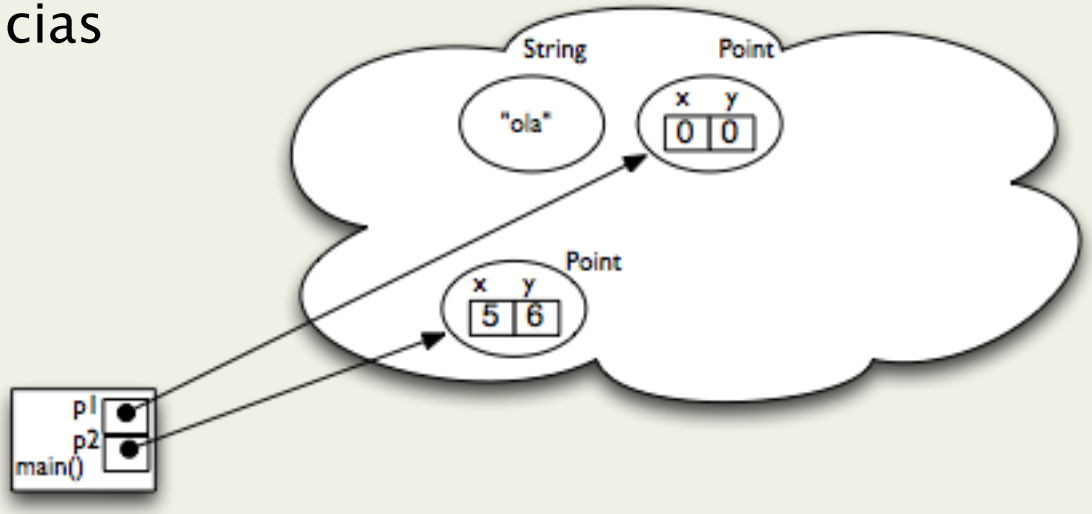
# Aplicações: Alocação de memória em Java

- Anteriormente vimos como a JVM trata de arranjar espaço para as variáveis locais de um método que foi invocada e as coloca na ***Java stack***
- Existe outra tipo de memória disponível para os dados do programa, a que se chama ***memory heap***
- A memória do ***heap*** está dividida em *blocos* (zonas de memória contíguas). A gestão de blocos livres de memória pode ser feita com uma fila.



# Aplicações: Alocação de memória em Java

- Alocação de memória no ***heap***
  - sempre que se faz uso do operador **new** é criado dinamicamente um objecto cuja existência não irá depender da terminação do método em que foi definido e que fica guardado no ***heap***
  - sempre que é executado o ***garbage collector*** é libertada a memória onde estão guardados objectos para os quais não existem referências



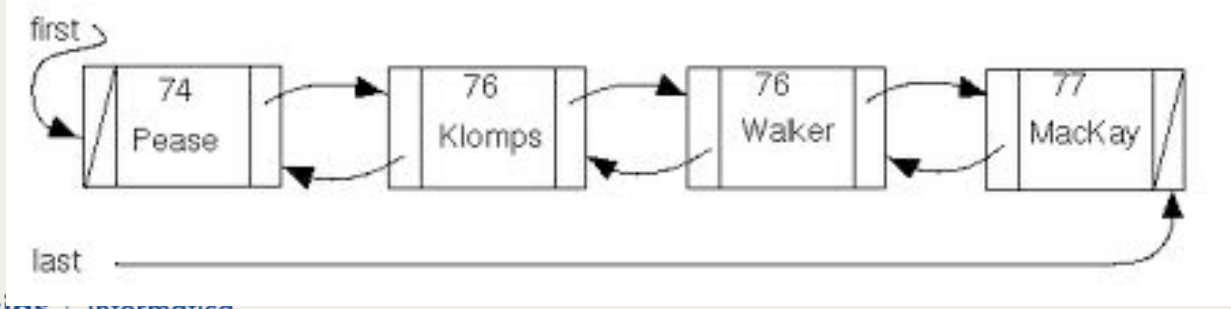
# Deque (*Double-Ended Queue*)

Uma fila de elementos que permite inserir e remover de ambos os extremos

- **make**: constrói fila sem nenhum elemento
- **addFirst**: insere no início da fila um elemento dado
- **addLast**: insere no fim da fila um elemento dado
- **removeFirst**: retira o elemento que está no início da fila
- **removeLast**: retira o elemento que está no fim da fila
- **getFirst**: indica o elemento no início da fila
- **getLast:** indica o elemento no fim da fila
- **size**: indica o número de elementos na fila

# Deque: Implementação

- A utilização de uma **lista simplesmente ligada** é ineficiente
- Uma estrutura de dados mais adequada é uma **lista duplamente ligada**:
  - cada nó tem três coisas:
    - um elemento *item*
    - uma referência *next* para o próximo nó
    - uma referência *prev* para o nó anterior nó
  - manter em *first* uma referência para o primeiro nó da lista ligada

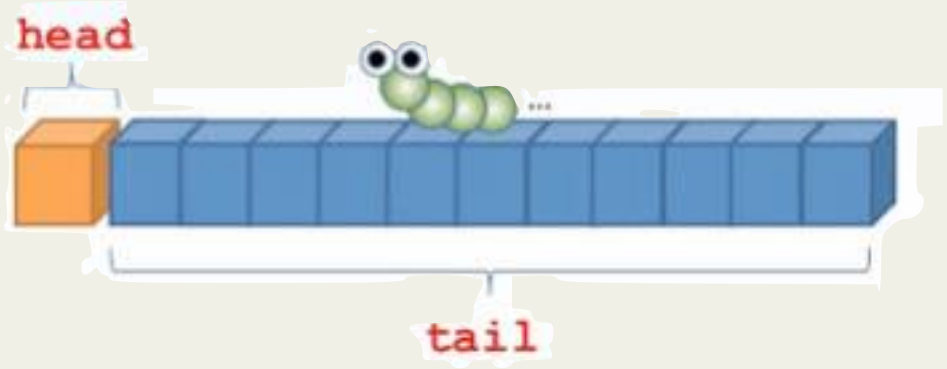



da lista

# List (na terminologia ADT)

Na terminologia standard de ADTs, as listas têm as seguintes operações

- **make**: constrói lista sem nenhum elemento
- **addFirst** juntar um elemento no inicio
- **head** determina o primeiro elemento da lista
- **tail** a lista com todos os elementos excepto a cabeça
- **isEmpty** que determina se a lista é vazia



# List (na terminologia Java)

Na terminologia Java as listas **suportam o acesso por índice** (***random access***) através das seguintes operações

- **make**: constrói lista sem nenhum elemento
- **get**: indica o elemento que está na posição dada
- **set**: coloca na posição indicada o elemento fornecido
- **add**: insere na posição indicada o elemento fornecido
- **remove**: retira da lista o elemento na posição dada
- **size**: indica o tamanho da lista

# List: Implementações (API)

```
interface List<E> {
   /**
   * @param i The index
   * @return The element of the list with index i
   * @requires 0<= i && i< size()
    */
   E get(int i);
   /** Replace with eand return the element at index i
   * @param i The index
   * @param e The element
   * @requires 0<= i && i< size()
    */
   void set(int i, E element);
   /** Insert e  into the list to have index i
   * @param i The index
   * @param e The element
   * @requires 0<= i && i<= size()
    */
   void add(int i, E element);
   ...
}
```

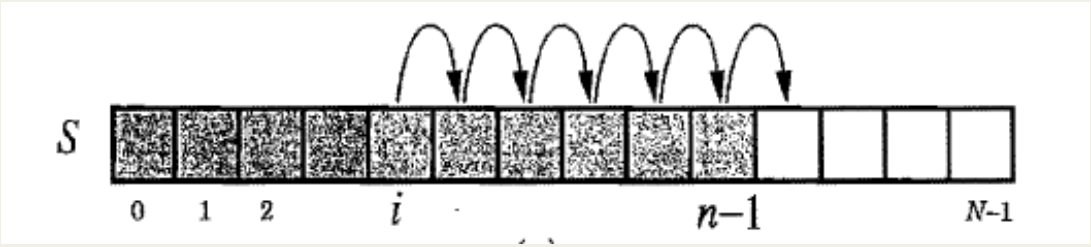
# List: Implementações (API)

```
interface List<E> {
   ...

   /** Removes from the list the element at index i
   * @param i The index
   * @requires 0<= i && i< size()
    */
   void remove(int i);
   /**
   * @return the size of the list
    */
   int size();
}
```

# List: Implementação com vector

# List: Implementação

- Implementação recorrendo **a um vector**
  - indice i do vector guarda i-ésimo elemento da lista
  - as operações `add(i,e)` e `remove(i)` obrigam a um deslocamento dos elementos nas posições contíguas

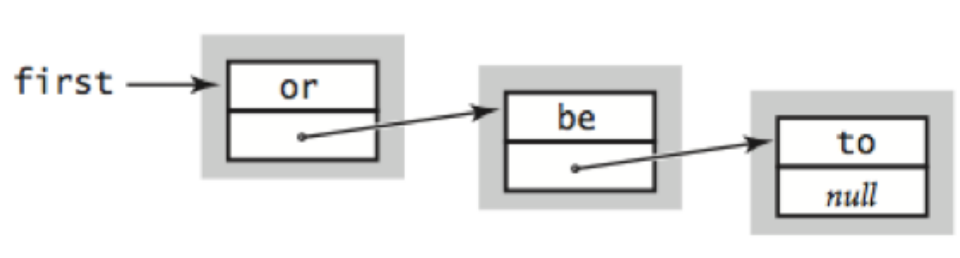



| Method | Time |
|---|---|
| get | O(1) |
| set | O(1) |
| add | O(n) |
| remove | O(n) |
| size | O(1) |

  - de forma a não restringir a inserção de elementos na lista, este **vector é dinâmico** (i.e., muda-se para um vector maior ou mais pequeno sempre que for conveniente)

# List: Implementação

- Implementação recorrendo a uma lista ligada
  - nó i da lista guarda o  i-ésimo elemento da lista




| Method | Time |
|---|---|
| get | O(n) |
| set | O(n) |
| add | O(n) |
| remove | O(n) |
| size | O(1) |

## Iteradores

- Uma computação típica sobre várias colecções de elementos, nomeadamente as listas, é percorrer os seus elementos, um de cada vez, por uma dada ordem
- Este tipo de computação pode ser abstraída através de um **iterador**, fazendo a classe **iterável**
- Tipos iteráveis, à semelhança dos vectores, podem ser usados em ***for-each***

```
int[] v = {5, 7, 9, 21, 3, 21};
int count = 0;
for (int x: v) {
   count += x;
}
```

# Interfaces Iterable e Iterator (java.util) no Java 8

```
interface Iterable<T> {
   //Returns an iterator over a set of elements of type T.
   Iterator<T> iterator();
}
```

```
/** T - the type of elements returned by this iterator */
interface Iterator<T> {

   //Returns true if the iteration has more elements.
   boolean hasNext();

   //Returns the next element in the iteration.
   T next();

   //Removes from the underlying collection the last
   //element returned by the iterator
   //(optional operation).
   default void remove();

   //Performs the given action for each remaining element
   default void forEachRemaining(Consumer<? super T> action);
}
```

# Iteradores

A iteração pode ser abstraída através de um **iterador**, fazendo a classe **iterável**

- ou seja, fazer a classe implementar o interface `Iterable`

```
class LinkedList<E> implements Iterable<E>{
   ...
   public Iterator<E> iterator(){
      return new ListIterator();
   }
   ...
}
```

# Iteradores e ciclos for-each

O ciclo ***for-each*** é aplicável a objectos de tipos iteráveis

```
LinkedList<String> list;
...
for (String x: list) {
   System.out.println(x);
}
```

# Iteradores e ciclos for-each

O ciclo ***for-each*** é aplicável a objectos de tipos iteráveis

```
LinkedList<String> list;
...
for (String x: list) {
   System.out.println(x);
}
```

```
LinkedList<String> list;
...
Iterator<String> i = list.iterator();
while (i.hasNext()) {
   String x = i.next();
   System.out.println(x);
}
```

## Implementação de Iteradores

- Em Java, a implementação dos iteradores deve ser feita através de uma classe interna e privada

- Por razões de eficiência em regra geral os iteradores não trabalham sobre uma cópia da estrutura mas sobre a estrutura original, recorrendo a cursores que sinalizam o ponto em que se vai na iteração
- A iteração e modificação concorrente de uma estrutura é desaconselhada

# List Iterator

```
public class LinkedList<E> implements Iterable<E>{
    //the first node in the list
    private Node<E> first;
      ...
    //private inner class
   private class ListIterator implements Iterator<E>{
        private Node<E> current;
        private ListIterator() {
            current = first;
        }
        public boolean hasNext() {
            return current != null;
        }
       //@requires hasNext();
        public E next() {
            E result = current.item;
            current = current.next;
            return result;
        }
    }
}
```

# Interface Iterator

Na verdade o contrato do método `next` no `java.util.Iterator` não impõe pré-condições aos seus clientes e ao invés exige que quem implementa este interface envie a excepção `NoSuchElementException` quando é este método é executado num estado em que `hasNext` é falso

### next

`E next()`

Returns the next element in the iteration.

**Returns:**

the next element in the iteration

**Throws:**

`NoSuchElementException` - if the iteration has no more elements

# List Iterator Corrigido

```
public class LinkedList<E> implements Iterable<E>{
    //the first node in the list
     private Node<E> first;
       ...
    //private inner class
   private class ListIterator implements Iterator<E>{
        private Node<E> current;
        private ListIterator() {
            current = first;
        }
        public boolean hasNext() {
            return current != null;
        }
        public E next() {
           if (current==null)
              throw new NoSuchElementException();
            E result = current.item;
            current = current.next;
            return result;
        }
    }
```