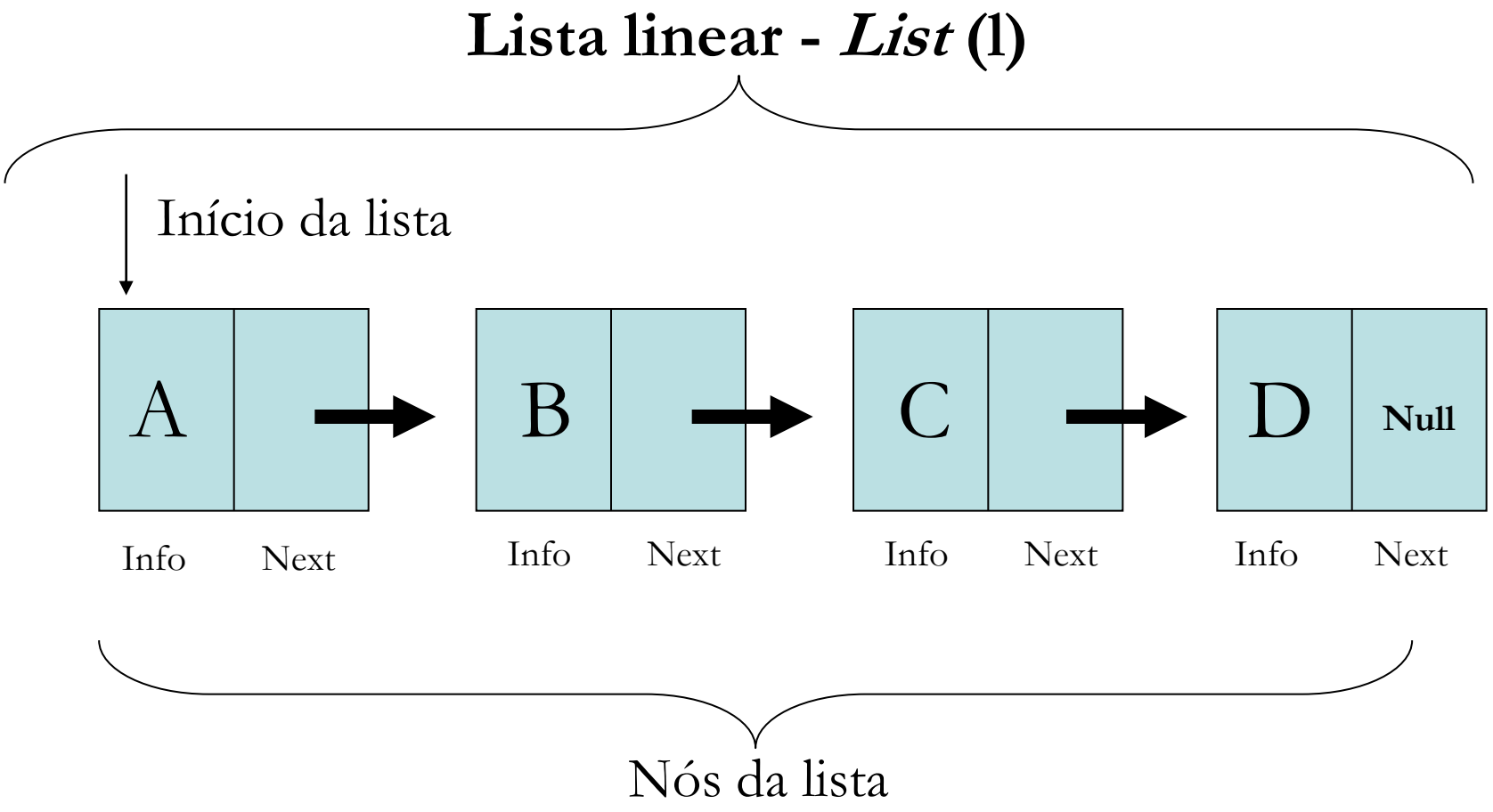
# Tipos abstratos de dados
# Listas

Prof. Tiago Massoni
Prof. Fernando Buarque
Prof. Byron Leite

Engenharia da Computação

Poli - UPE

# Lista - intuição



# Listas ligadas (encadeadas)

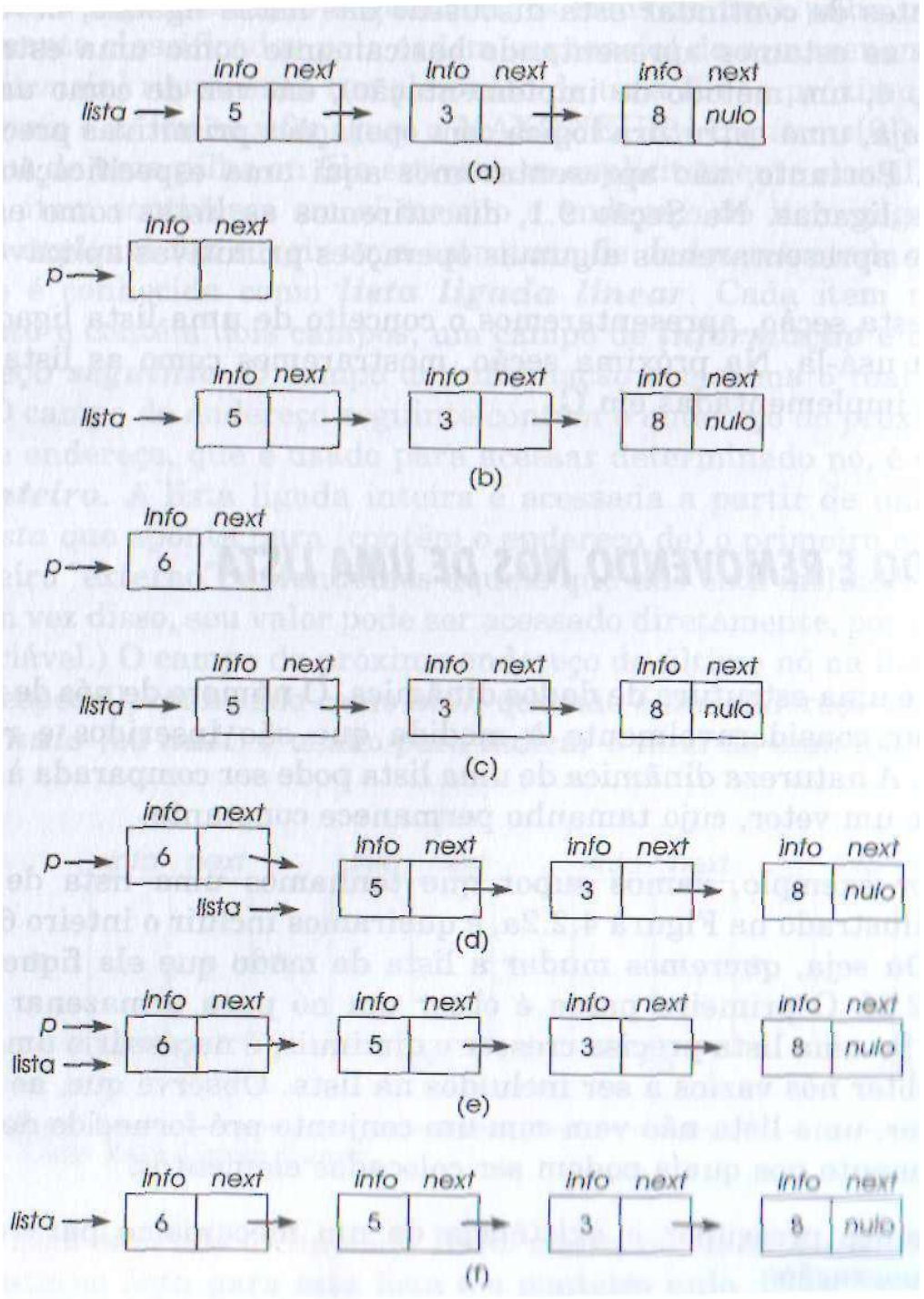
- Não são armazenadas de forma contígua
  - Nós não estão necessariamente em sequência na memória
- printList e find: começa a procura a partir do começo da lista
  - Uso do next de cada nó
- findKth mais ineficiente que array
- insert, remove: mais eficientes
  - Insert: uma criação de objeto e duas mudanças de referências
  - Remove: uma mudanças de referências

# Implementação - nós de uma lista ligada

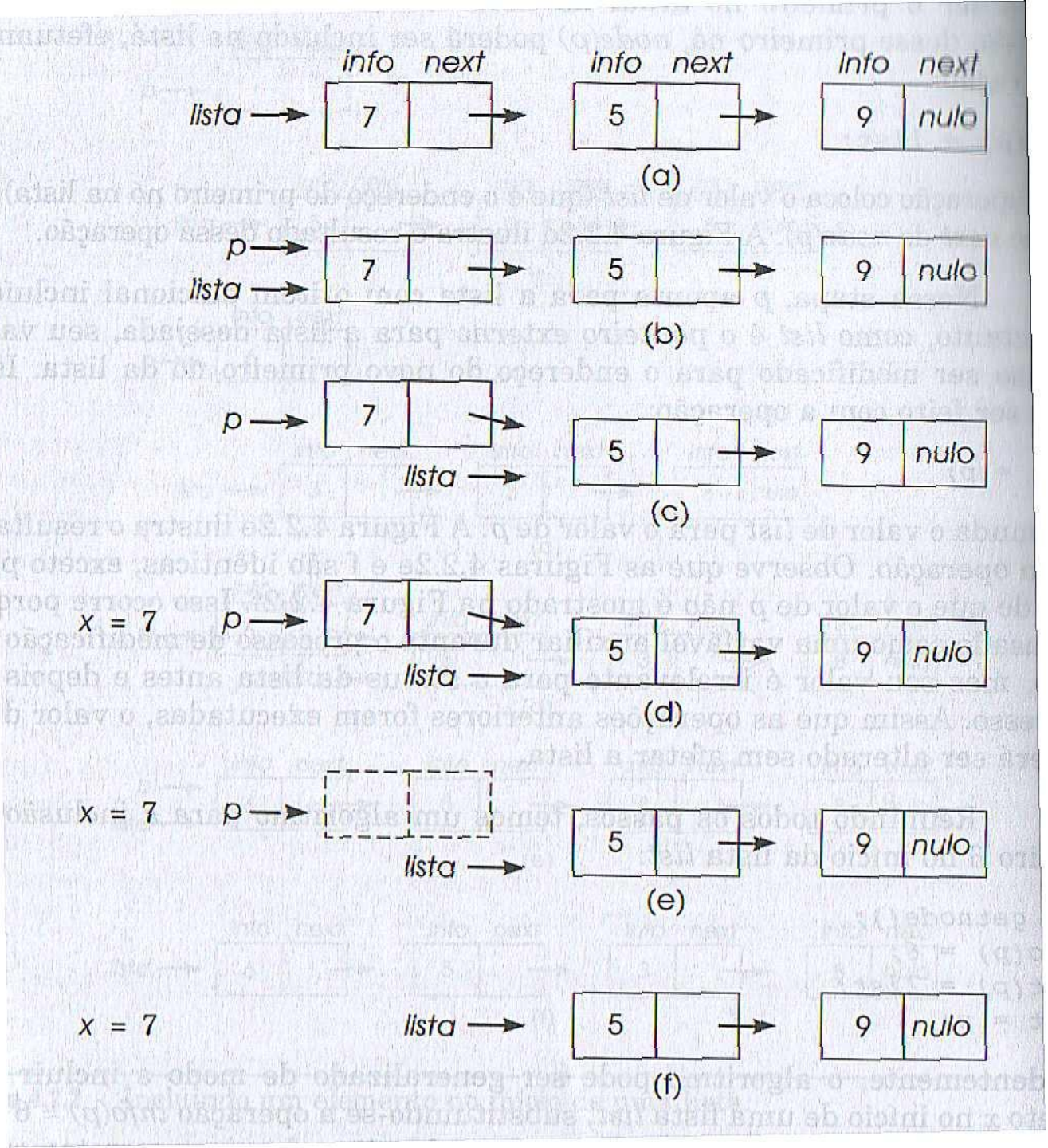
```
public class Node {
      private Object info;
      private Node next;

      public Node(Object el){
            this.info = el;
      }
      public Node(Object el, Node next){
            this.next = next;
      }
      //metodos get e set
}
```

# Inserção de nó no início



# Remoção de nó do início



# Detalhes de implementação

- Ao inserir ou remover do início temos que tratar da referência inicial da lista
  - Casos especiais
  - Solução – colocar um nó cabeçalho (header) na lista – não possui conteúdo
    - Evita casos especiais
- Ao remover um nó, temos que ter acesso ao seu nó anterior
  - Solução: findPrevious(Object x)

# Implementação - Lista (com header)

```
public class LinkedList {
  private Node header;

  public LinkedList( ) {
    header = new Node(null);
  }

  public boolean isEmpty( ){..}

  public void makeEmpty( ){..}

  public ListIterator zeroth( ){..}

  public ListIterator first( ){..}
```

# Implementação - Lista (com header)

```
 public void insert(Object x, ListIterator p){
  if (p!=null && !p.isPastEnd())
    p.setNext(new Node(x,p.getNext());
  ...
 }

 public ListIterator find(Object x){..}

 public ListIterator findPrevious(Object x){..}

 public void remove(Object x){..} //usa findPrevious

 public void printList(){..}

}
```

# Iterator de Lista

- Vamos usar um objeto para representar a noção de posição
  - Não vamos usar inteiros, para deixar a implementação mais flexível
- Também separaremos neste objeto a navegação na lista por posições
  - Separação entre a implementação da lista e as operações de navegação
- Este objeto – Iterator (Iterador)
  - Oferece navegação a partir de uma dada posição

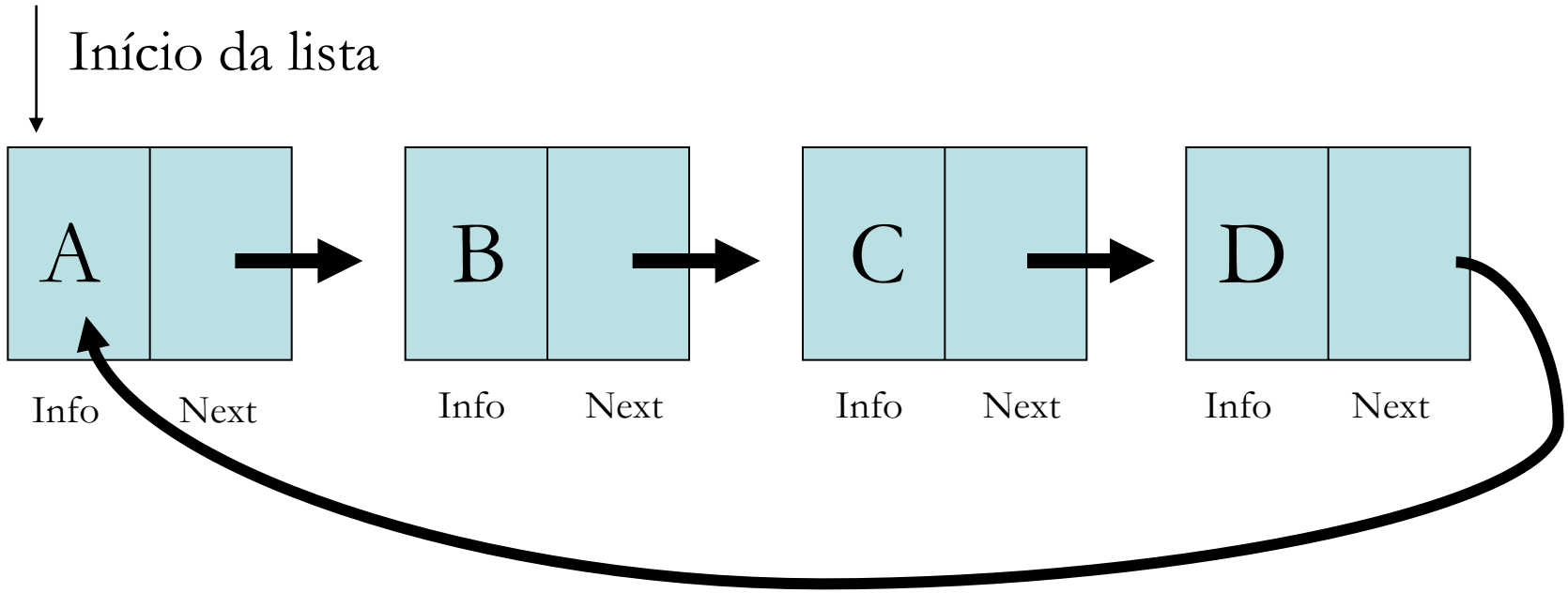
# Implementação - Iterator

```
public class ListIterator {
    Node current;
    ListIterator(Node n){
      current= n;
    }
    public boolean isPastEnd() {
      return (current == null);
    }
    public Object retrieve() {
      if (isPastEnd())
            return null;
      else  current.getElement();
    }
    public void advance(){
      if(!isPastEnd())
         current = current.next;
    }
    public Node getNext(){...}
    public Node setNext(){...}
}
```

# Limitação das listas

- Dificuldade para do final  da lista retornar ao começo
  - Listas circulares
- Dificuldade de acessar o elemento anterior
  - Listas duplamente ligadas (encadeadas)

# Listas ligadas circulares

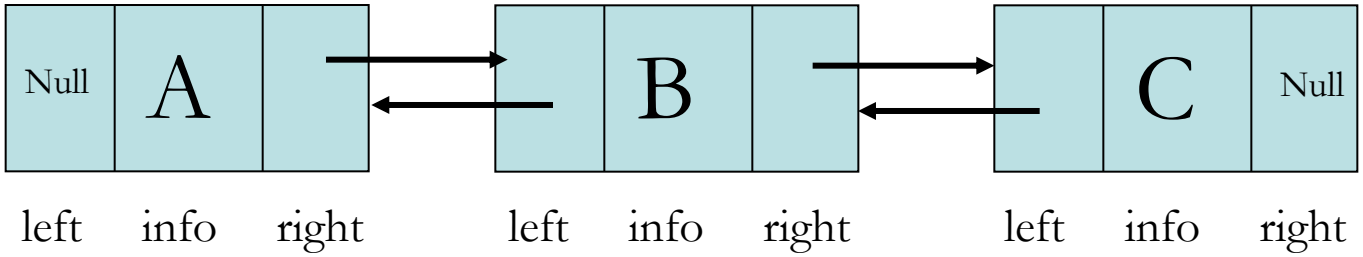


# Listas ligadas circulares

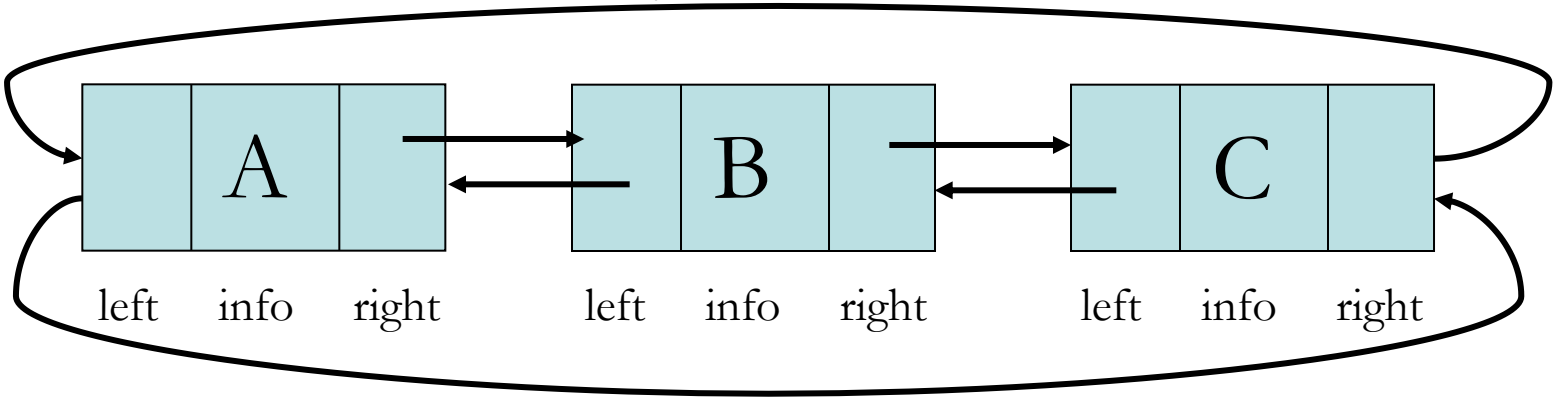
- Se existir nó cabeçalho, usá-lo com cuidado
  - Último nó da lista referencia header
- Operações primitivas – alguns testes são afetados
  - insert(x,p) idêntico às listas simples
  - remove(x)?

# Listas duplamente ligadas

- Lista linear duplamente ligada:




- Lista circular duplamente ligada:



# Listas duplamente ligadas

```
public class DoubleNode {
      private Object info;
      private Node prev;
      private Node next;
      ...
}
```

- Custo aumenta: atributo extra que armazena ref. para anterior
  - insert e remove também fazem mais mudanças
- Simplifica remoção
  - Não precisa mais usar findPrevious