Listas Lineares Ordenadas

Introdução

- Na apresentação anterior comentamos brevemente que os algoritmos de listas podem ser melhorados em termos de eficiência e simplicidade, fazendo-se uso do elemento **sentinela**
- Esta técnica simples é válida para ambas implementações (contígua e encadeada) tanto para **listas lineares** como **listas lineares ordenadas**

Organização

- Definição do ADT Lista Ordenada
- Especificação
  - Operações sobre o ADT Lista Ordenada, utilizando pré- e pós-condições
- Implementação
  - Estática (contígua, deixada como exercício)
  - Dinâmica (encadeada)

Definição

- Uma **lista linear ordenada** é uma lista linear de **n** elementos **[a_1, a_2, ..., a_i, ..., a_n]**
  - O primeiro elemento da lista é **a_1**
  - O segundo elemento da lista é **a_2**
  - ...
  - O **i**-ésimo elemento da lista é **a_i**
  - ...
  - O último elemento da lista é **a_n**
- Na qual há uma relação de ordem (**<**, **<=**, **>**, **>=**) entre os elementos:
  - Ordem crescente: a_1 **<** a_2 **<** ... **<** a_{n-1} **<** a_n ou a_1 **<=** a_2 **<=** ... **<=** a_{n-1} **<=** a_n
  - Ordem decrescente: a_1 **>** a_2 > ... **>** a_{n-1} **>** a_n ou a_1 **>=** a_2 >= ... **>=** a_{n-1} **>=** a_n
- Trataremos de listas em ordem crescente, sem nenhum prejuízo de generalidade

Especificação

- Operações:
  - Criação
  - Destruição
  - Status
  - Operações Básicas
  - Outras Operações

Criação

OrderedList::OrderedList();

- *pré-condição:* nenhuma
- *pós-condição:* Lista é criada e iniciada como vazia

Destruição

OrderedList::~OrderedList();

- *pré-condição:* Lista já tenha sido criada
- *pós-condição:* Lista é destruída, liberando espaço ocupado pelo seus elementos

Status

bool OrderedList::Empty();

- *pré-condição:* Lista já tenha sido criada
- *pós-condição:* função retorna **true** se a Lista está vazia; **false** caso contrário

Status

bool OrderedList::Full();

- *pré-condição:* Lista já tenha sido criada
- *pós-condição:* função retorna **true** se a Lista está cheia; **false** caso contrário

Operações Básicas

void OrderedList::Insert(ListEntry x);

- *pré-condição:* Lista já tenha sido criada, não está cheia
- *pós-condição:* O item **x** é armazenado em ordem na Lista

O tipo **ListEntry** depende da aplicação e pode variar desde um simples caracter ou número até uma **struct** ou **class** com muitos campos

Operações Básicas

void OrderedList::Delete(ListEntry x);

- *pré-condição:* Lista já tenha sido criada, não está vazia e o item **x** pertence à lista
- *pós-condição:* O item **x** é removido da Lista, mantendo-a ordenada

Operações Básicas

int OrderedList::Search(ListEntry x);

- *pré-condição:* Lista já tenha sido criada
- *pós-condição:* Retorna a posição **p** da Lista em que **x** foi encontrado $1 \leq$ **p** $\leq$ **n**, onde **n** é o número de entradas na Lista; caso exista mais de um **x** na Lista, retorna o primeiro encontrado; retorna zero caso **x** não pertença à Lista

Outras Operações

void OrderedList::Clear();

- *pré-condição:* Lista já tenha sido criada
- *pós-condição:* todos os itens da Lista são descartados e ela torna-se uma Lista vazia

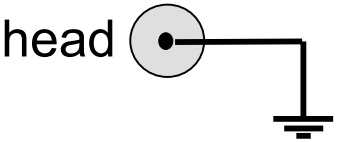
Outras Operações

int OrderedList::Size();

- *pré-condição:* Lista já tenha sido criada
- *pós-condição:* função retorna o número de itens na Lista

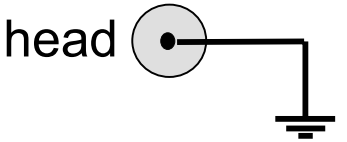
Implementação Encadeada

- Por razões de eficiência, introduziremos uma variável privada nos campos do objeto chamada **sentinel**, que será alocada no construtor e liberado no destruidor
- Além disso, ao invés de iniciarmos
  - head = NULL;

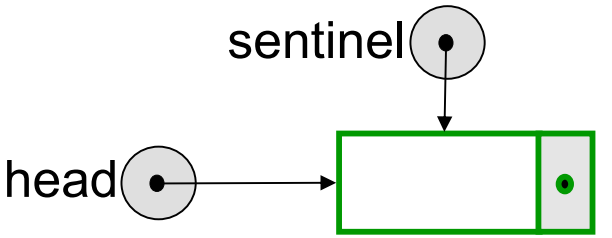


Implementação Encadeada

- Por razões de eficiência, introduziremos uma variável privada nos campos do objeto chamada **sentinel**, que será alocada no construtor e liberado no destruidor
- Além disso, ao invés de iniciarmos
  - head = NULL;

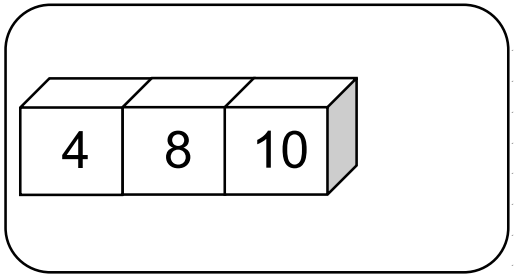


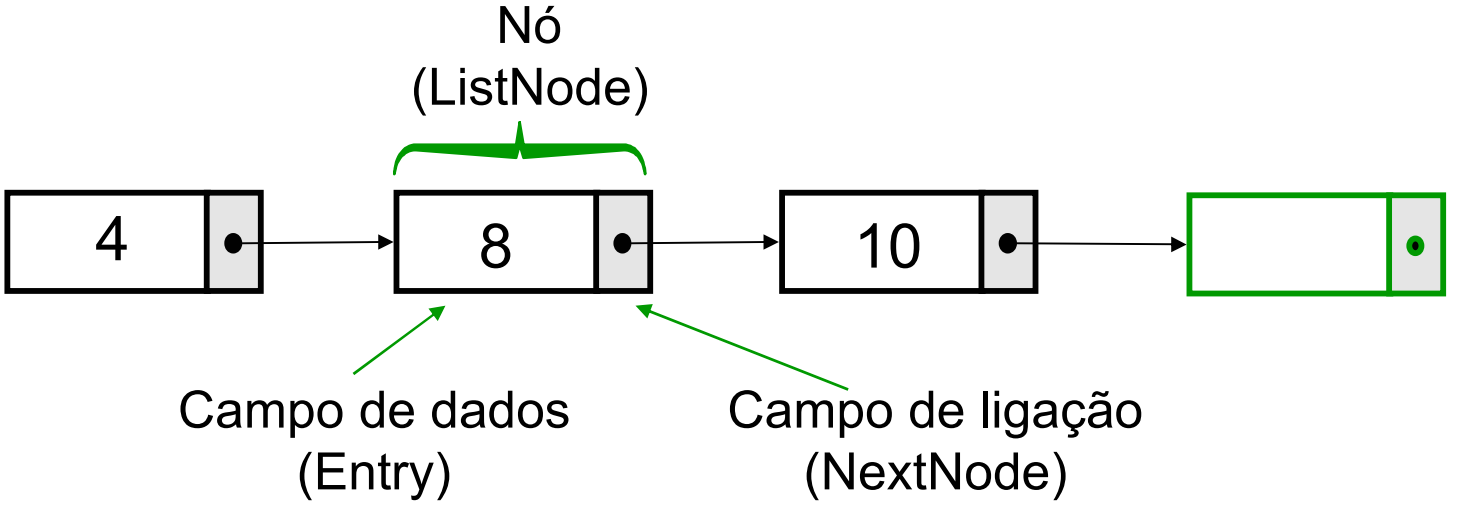
- Utilizaremos:
  - head = sentinel;



Implementação Encadeada

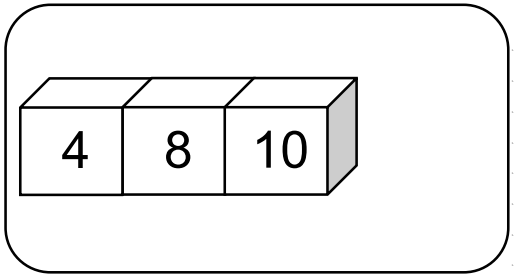
- As entradas de uma lista são colocadas em um estrutura (**ListNode**) que contém um campo com o valor existente na lista (**Entry**) e outro campo é um apontador para o próximo elemento na lista (**NextNode**)



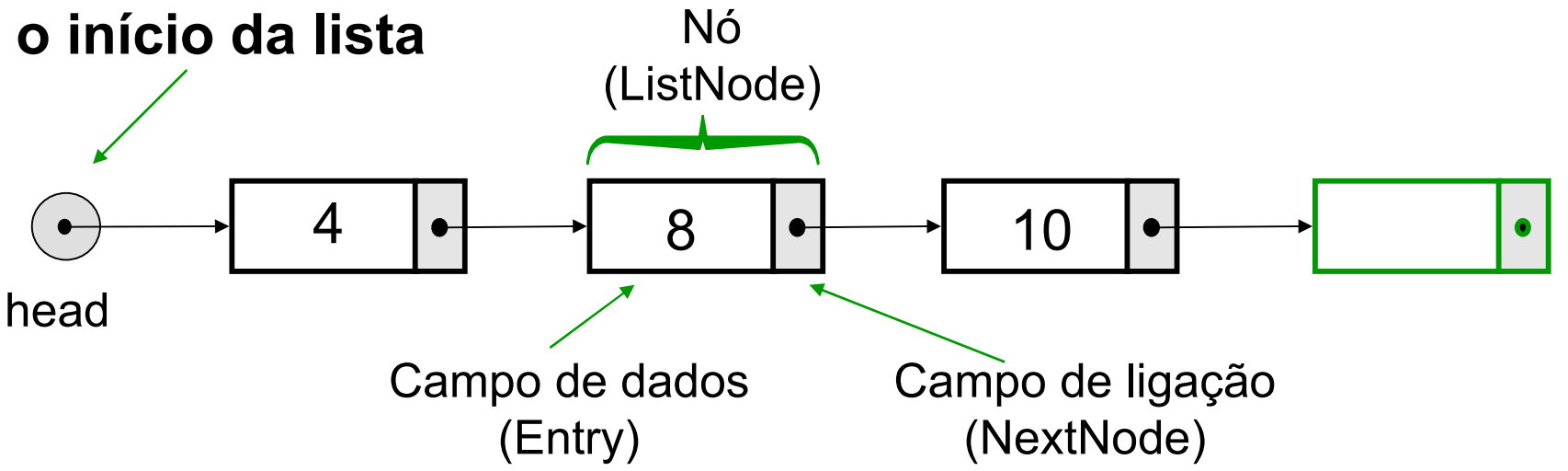


Implementação Encadeada

- Nós precisamos armazenar o início da lista...

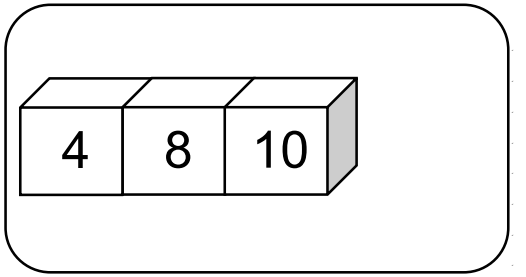


Um ponteiro armazena o início da lista

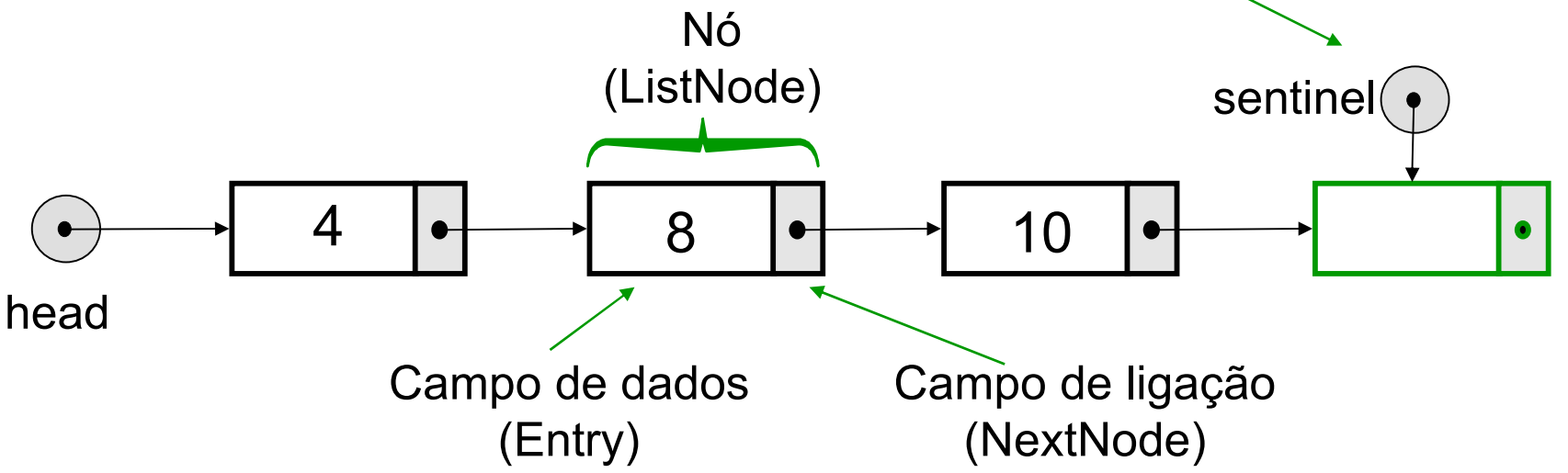


Implementação Encadeada

- Nós precisamos armazenar o início da lista e o sentinela...

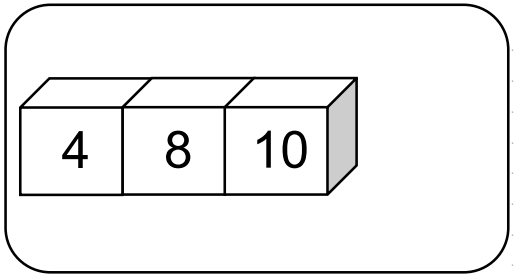


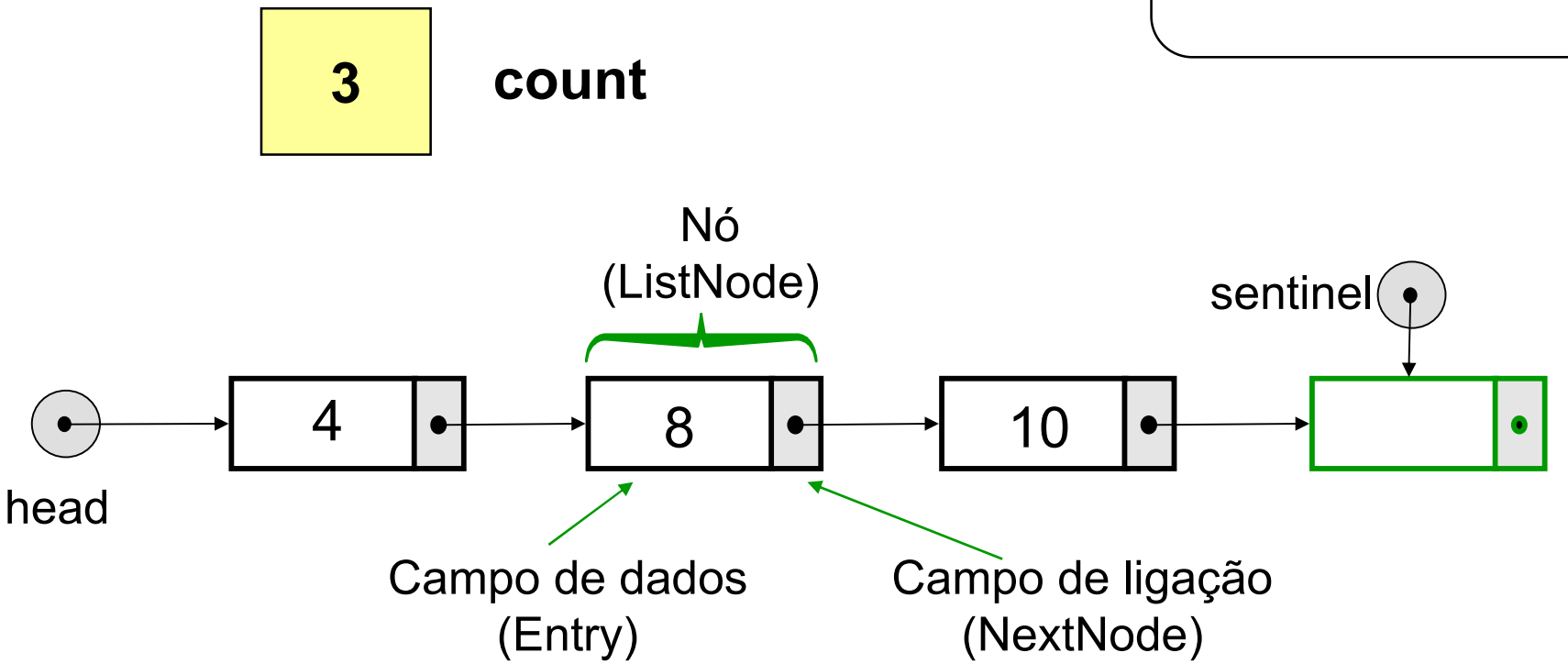
Um ponteiro armazena o elemento sentinela



Implementação Encadeada

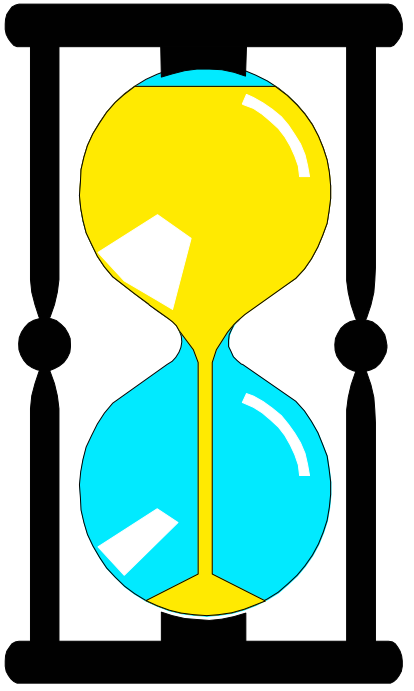
- ...e um contador que indica a quantidade de elementos na lista





Questão

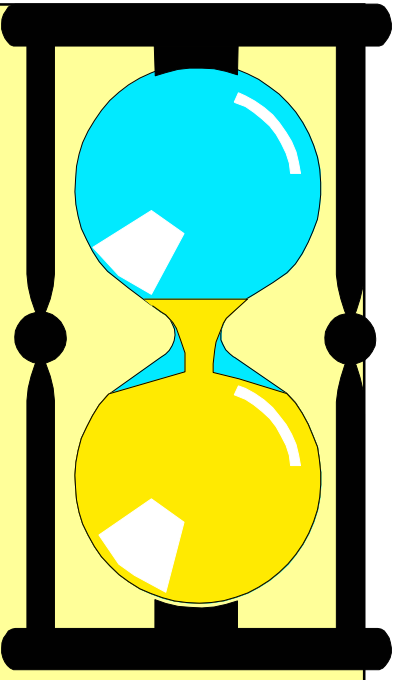
Utilize estas idéias para escrever uma declaração de tipo que poderia implementar uma lista encadeada. A declaração deve ser um objeto com dois campos de dados



Você tem 5 minutos
para escrever a declaração

Uma Solução

```
class OrderedList
{ public:
    OrderedList();
    ~OrderedList();
    void Insert(int x);
    void Delete(int x);
    int Search(int x);
    bool Empty();
    bool Full();
  private:
    // declaração de tipos
    struct ListNode
    { int Entry;                    // tipo de dado colocado na lista
      ListNode *NextNode;           // ligação para próximo elemento na lista
    };
    typedef ListNode *ListPointer;

    // declaração de campos
    ListPointer head, sentinel;   // início da lista e sentinela
    int count;                    // número de elementos
};
```



Uma Solução

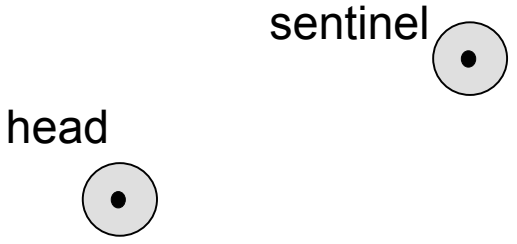
```
class OrderedList
{ public:
    OrderedList();
    ~OrderedList();
    void Insert(int x);
    void Delete(int x);
    int Search(int x);
    bool Empty();
    bool Full();
  private:
    // declaração de tipos
    struct ListNode
    { int Entry;                    // tipo de dado colocado na lista
      ListNode *NextNode;           // ligação para próximo elemento na lista
    };
    typedef ListNode *ListPointer;

    // declaração de campos
    ListPointer head, sentinel;   // início da lista e sentinela
    int count;                    // número de elementos
};
```

Observe que o tipo **ListEntry** nesse caso é um inteiro

Construtor

OrderedList::OrderedList()

A Lista deve iniciar vazia, cuja convenção é apontar para o elemento sentinela

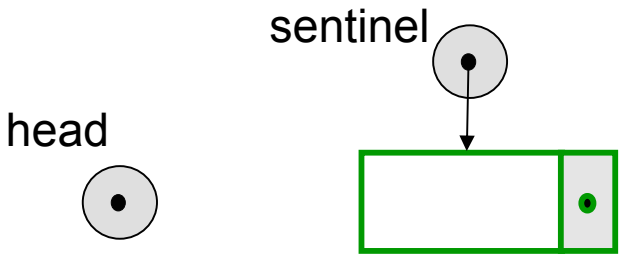



```
OrderedList::OrderedList()
{
    ...
}
```

Construtor

OrderedList::OrderedList()

A Lista deve iniciar vazia, cuja convenção é apontar para o elemento sentinela

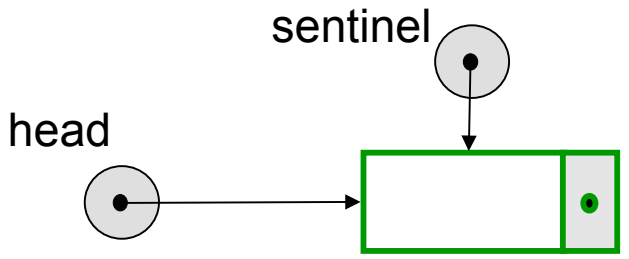



```
OrderedList::OrderedList()
{
    sentinel = new ListNode;
    ...
}
```

Construtor

OrderedList::OrderedList()

A Lista deve iniciar vazia, cuja convenção é apontar para o elemento sentinela

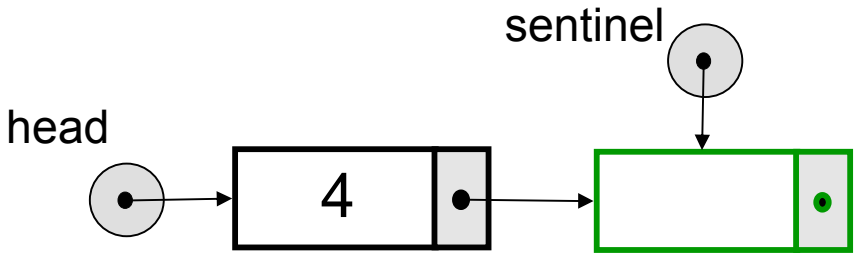



```
OrderedList::OrderedList()
{
    sentinel = new ListNode;
    head = sentinel;
    count = 0;
}
```

Destruidor

OrderedList::~OrderedList()

O destruidor deve retirar todos os elementos da lista enquanto ela não estiver vazia. Lembre-se que atribuir NULL a **head** não libera o espaço alocado anteriormente e que o elemento sentinela também deve ser liberado!

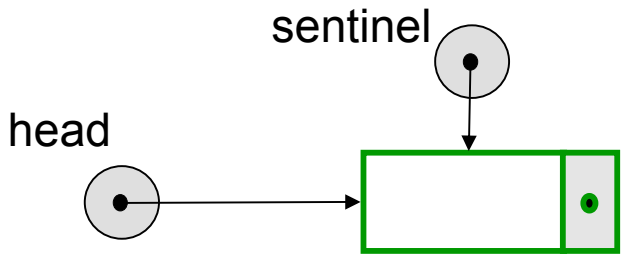



```
OrderedList::~OrderedList()
{ ListPointer q;

 while (head != sentinel)
  {  q = head;
     head = head->NextNode;
     delete q;
  }
  delete sentinel;
}
```

Status: Empty

bool OrderedList::Empty()

Lembre-se que a lista inicia vazia, com **head** = **sentinel**...

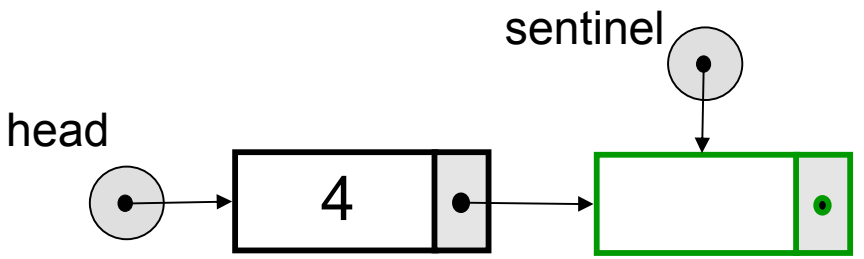



```
bool OrderedList::Empty()
{
    return (head == sentinel);
}
```

Status: Full

bool OrderedList::Full()

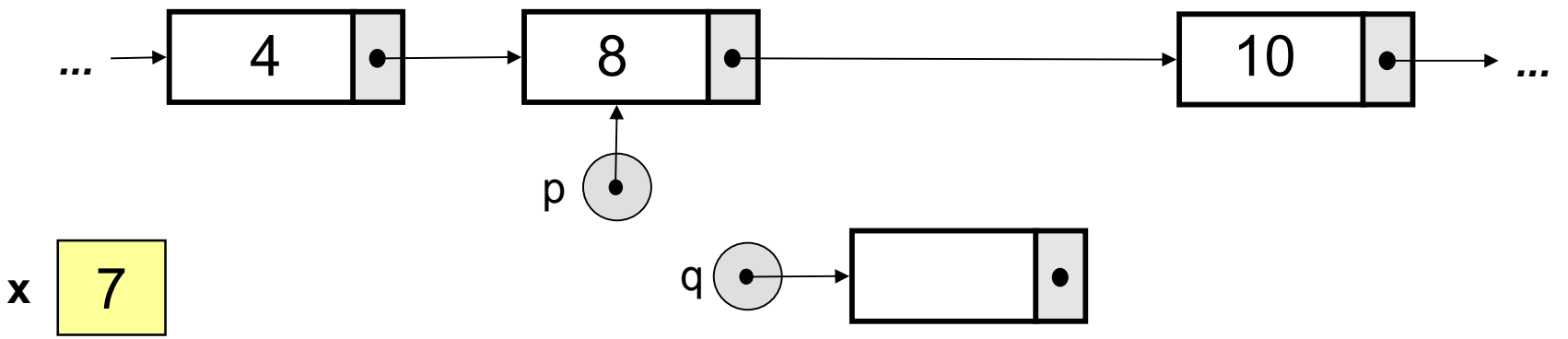
...e que não há limite quanto ao número máximo de elementos da lista




```
bool OrderedList::Full()
{
   return false;
}
```

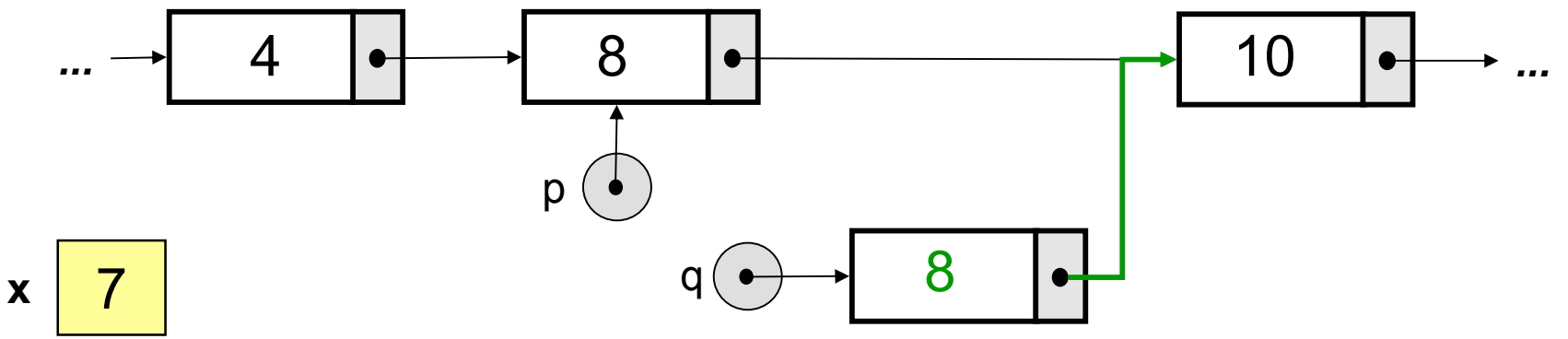
Inserção Antes de um Elemento

- A inserção de um elemento **x** designado por uma variável ponteiro **q** que deve ser inserido em uma lista *antes* do elemento apontado por **p**



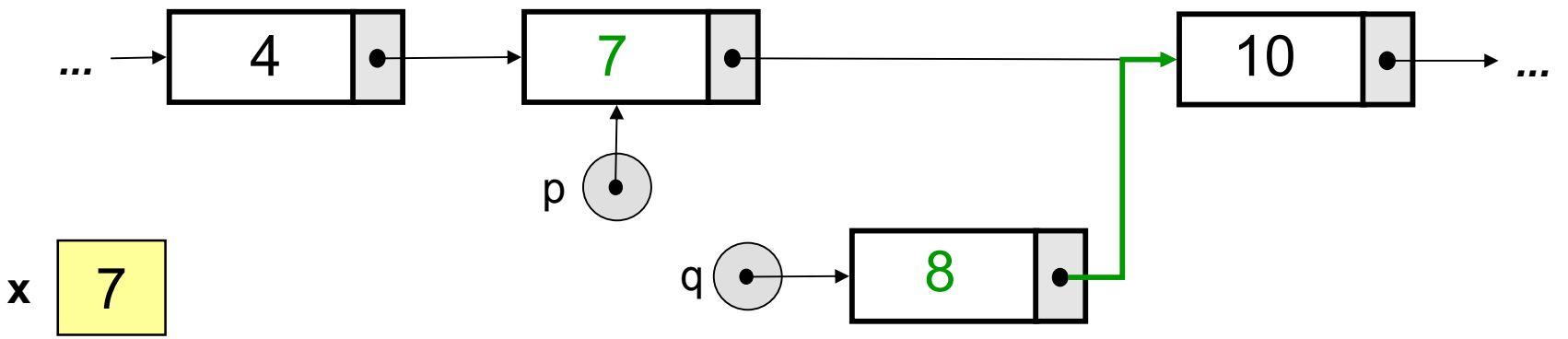
Inserção Antes de um Elemento

- A inserção de um elemento **x** designado por uma variável ponteiro **q** que deve ser inserido em uma lista *antes* do elemento apontado por **p**
  - *q = *p;



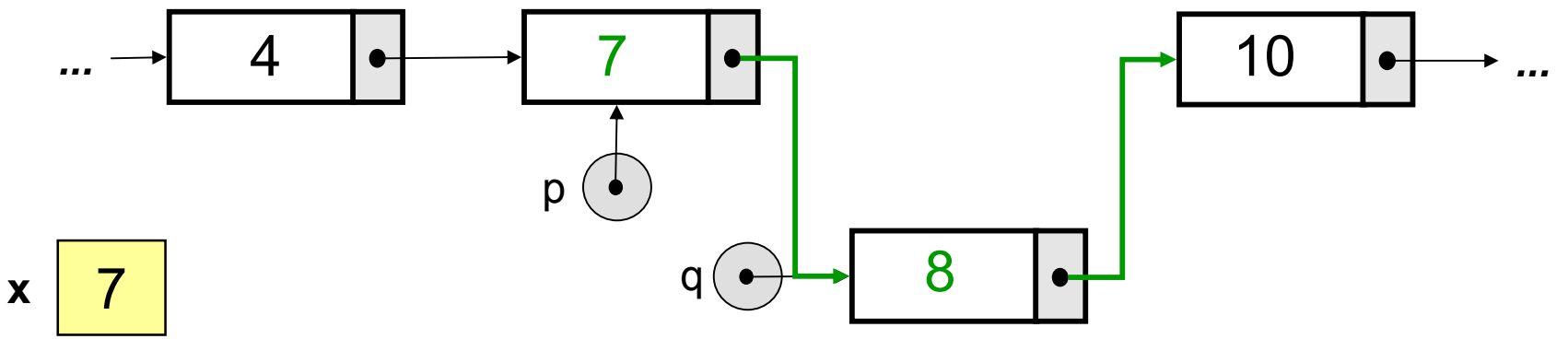
Inserção Antes de um Elemento

- A inserção de um elemento **x** designado por uma variável ponteiro **q** que deve ser inserido em uma lista *antes* do elemento apontado por **p**
  - *q = *p;
  - p->Entry = x;



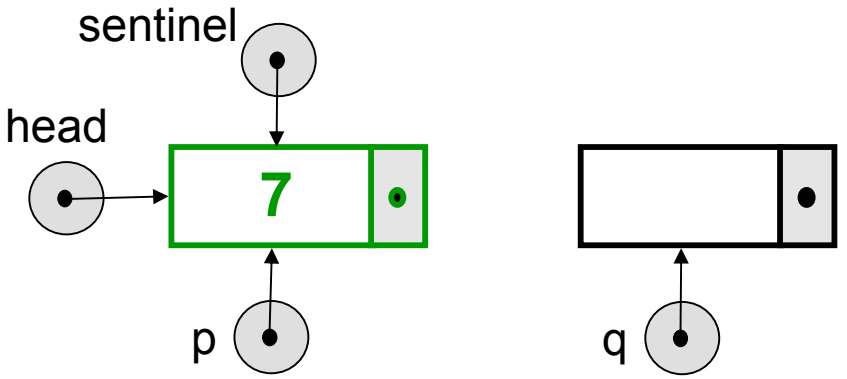
Inserção Antes de um Elemento

- A inserção de um elemento **x** designado por uma variável ponteiro **q** que deve ser inserido em uma lista *antes* do elemento apontado por **p**
  - *q = *p;
  - p->Entry = x;
  - p->NextNode = q;



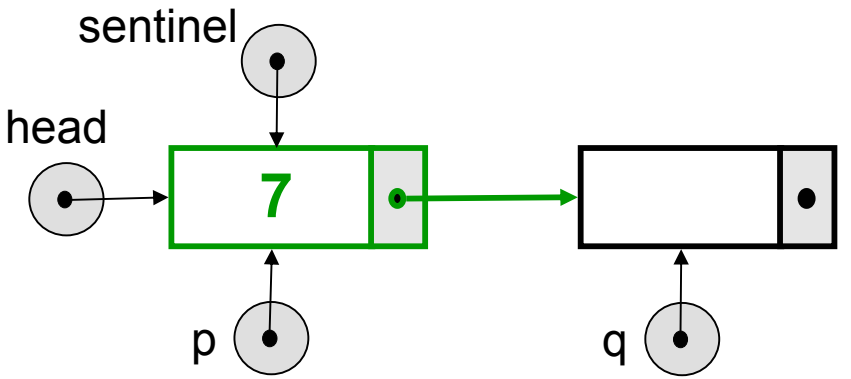
Operações Básicas: Insert

- Entretanto, se a inserção ocorrer no sentinela (no início...



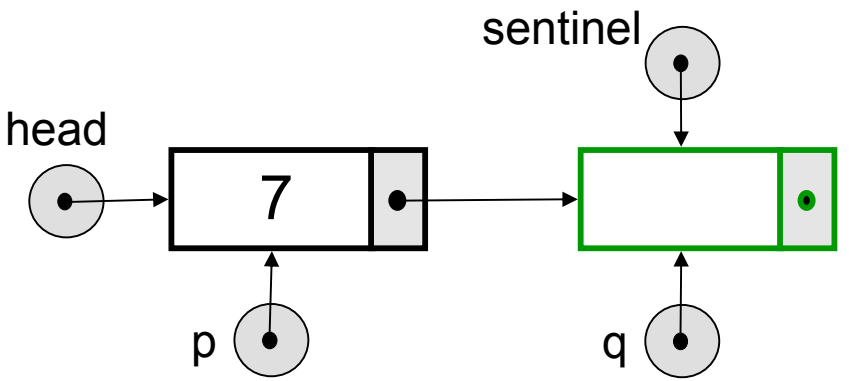
Operações Básicas: Insert

- Entretanto, se a inserção ocorrer no sentinela (no início...
  - p->NextNode = q;



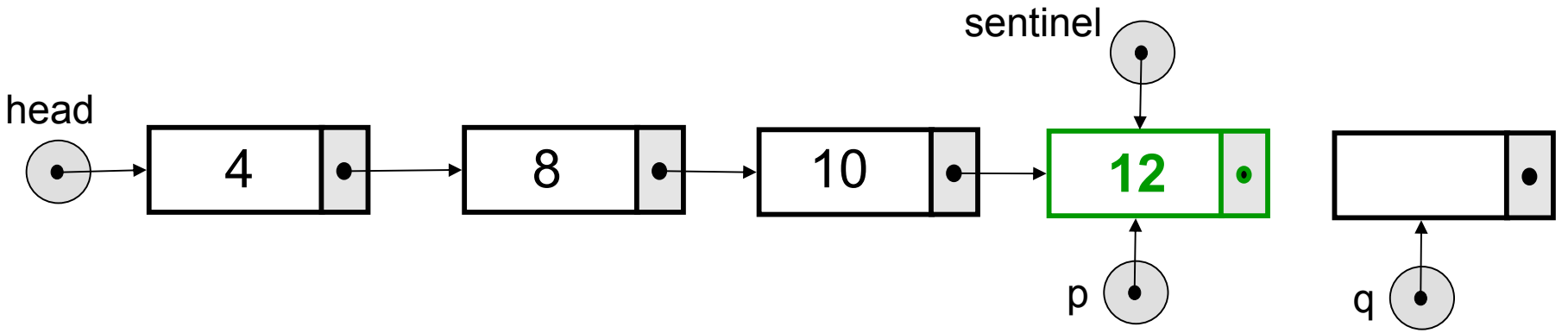
Operações Básicas: Insert

- Entretanto, se a inserção ocorrer no sentinela (no início...
  - p->NextNode = q;
  - sentinel = q;



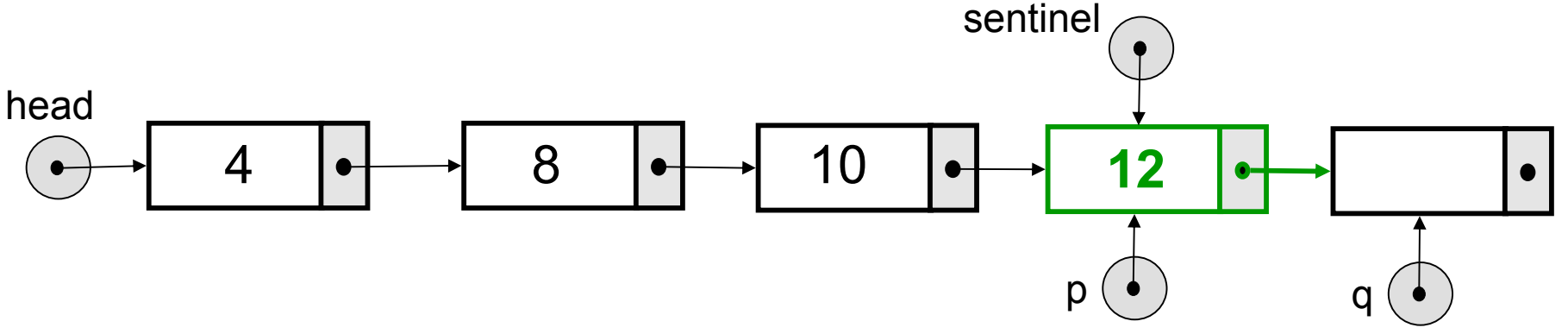
Operações Básicas: Insert

- Entretanto, se a inserção ocorrer no sentinela (no início ou no fim)
  - p->NextNode = q;
  - sentinel = q;



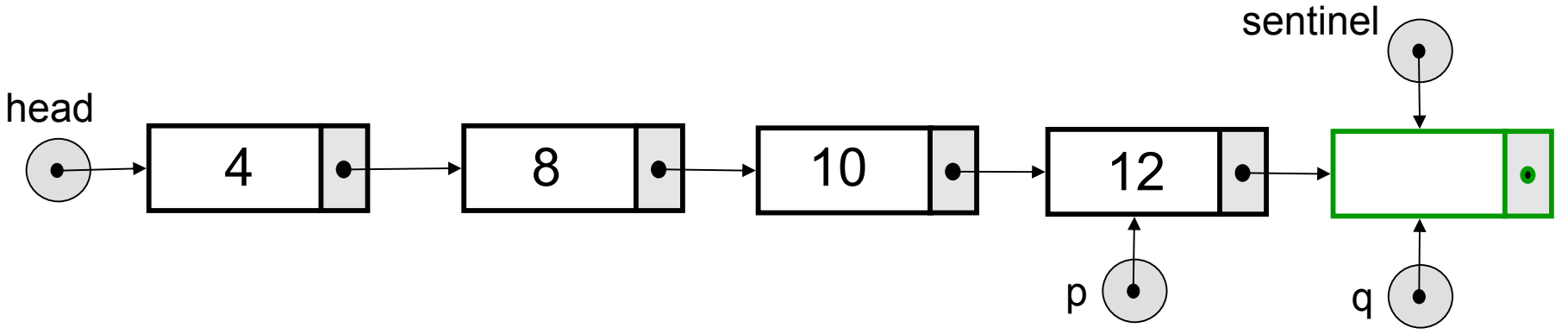
Operações Básicas: Insert

- Entretanto, se a inserção ocorrer no sentinela (no início ou no fim)
  - p->NextNode = q;
  - sentinel = q;



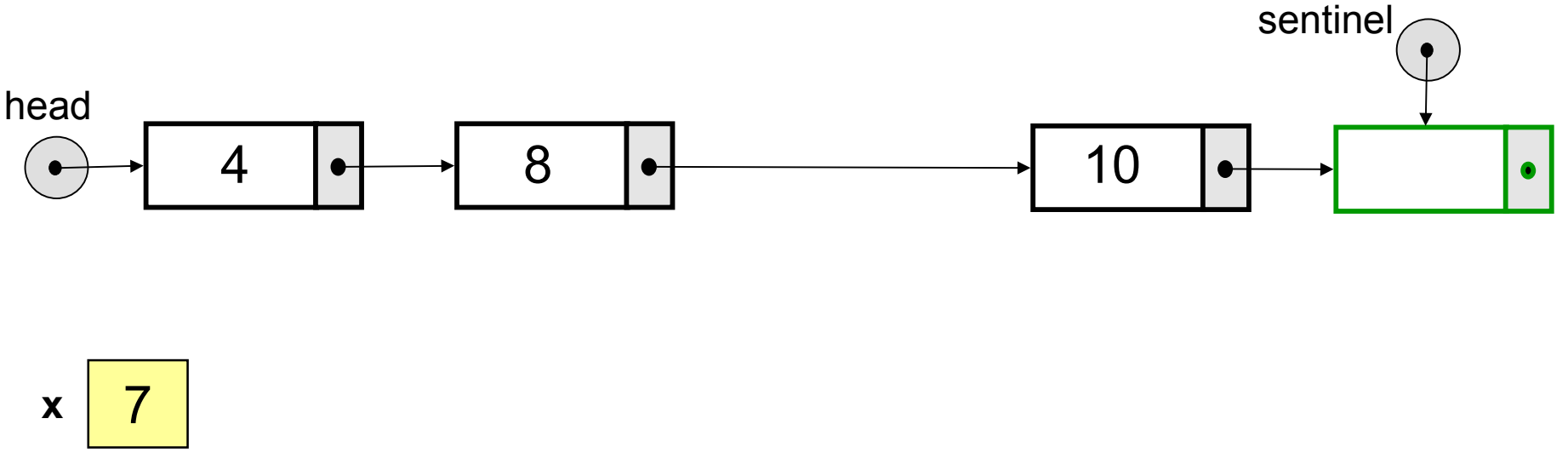
Operações Básicas: Insert

- Entretanto, se a inserção ocorrer no sentinela (no início ou no fim)
  - p->NextNode = q;
  - sentinel = q;



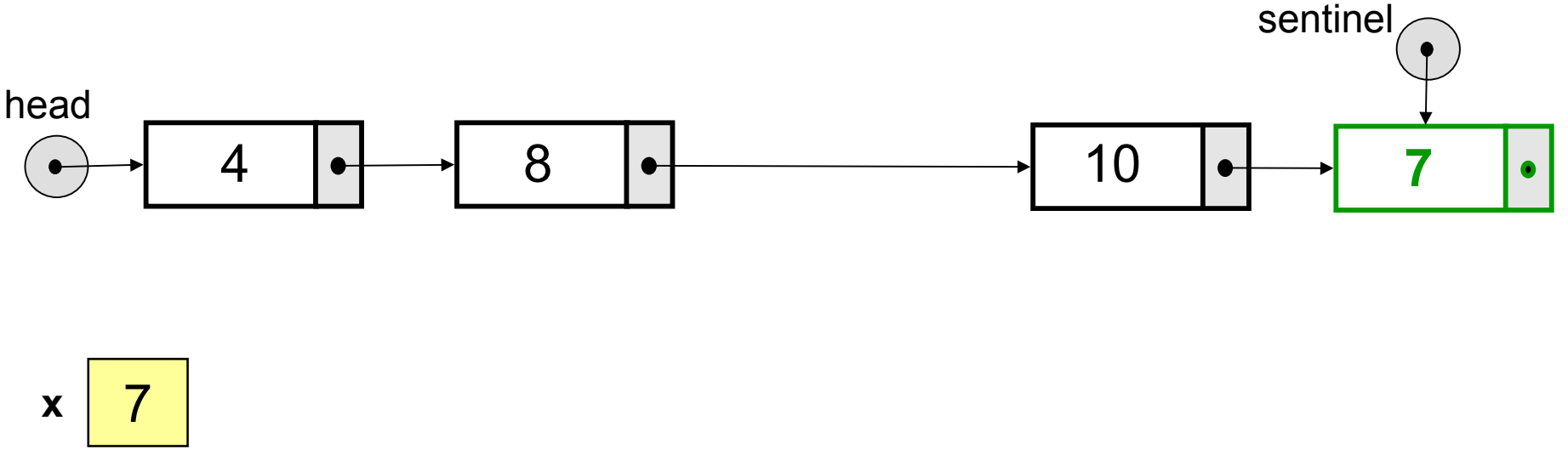
Ponto de Inserção

- Para encontrarmos a posição onde um novo elemento **x** deve ser inserido é simples, com o uso de sentinela



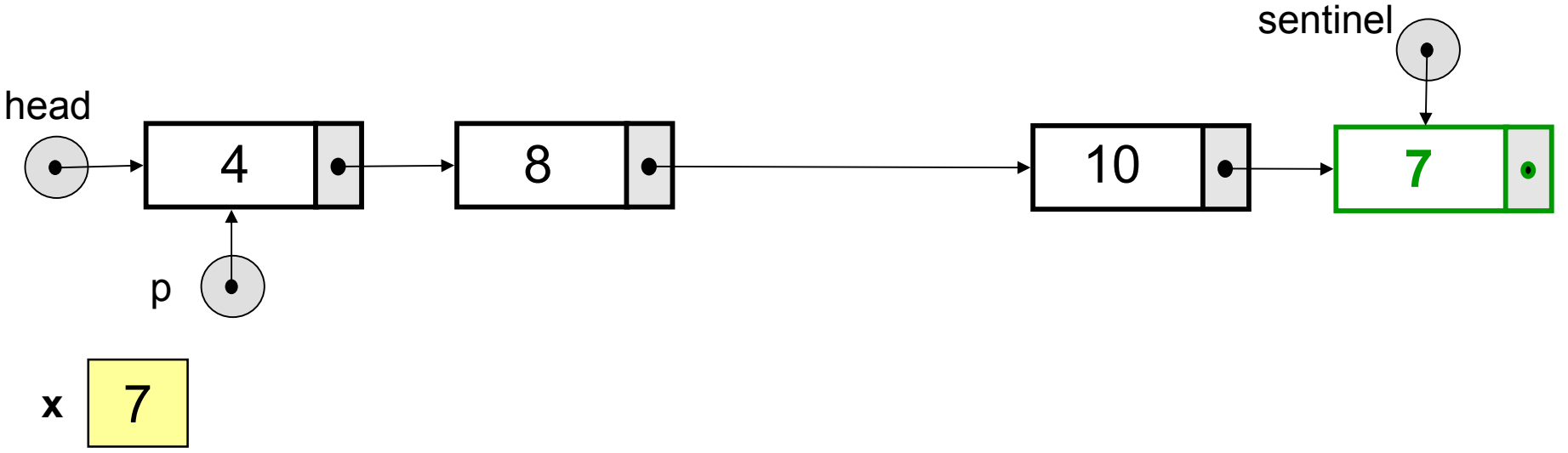
Ponto de Inserção

- Para encontrarmos a posição onde um novo elemento **x** deve ser inserido é simples, com o uso de sentinela
  - sentinel->Entry = x;



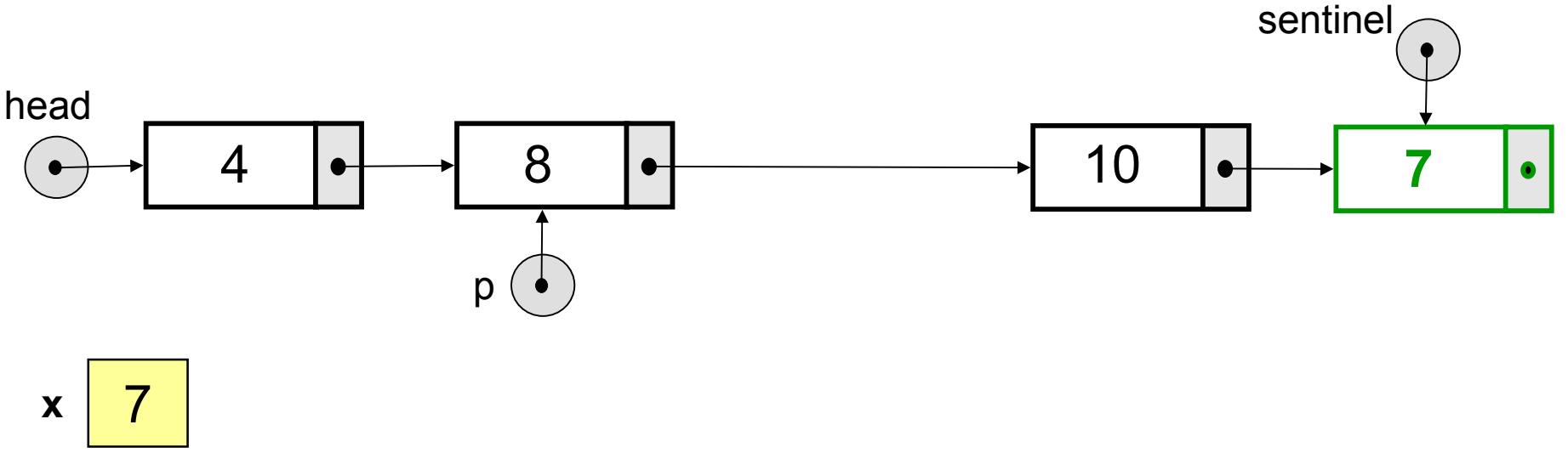
Ponto de Inserção

- Para encontrarmos a posição onde um novo elemento **x** deve ser inserido é simples, com o uso de sentinela
  - sentinel->Entry = x;
  - p = head;



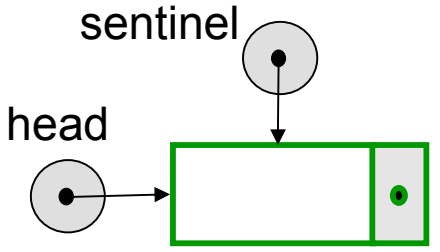
Ponto de Inserção

- Para encontrarmos a posição onde um novo elemento **x** deve ser inserido é simples, com o uso de sentinela
  - sentinel->Entry = x;
  - p = head;
  - while(p->Entry < x)
    - p = p->NextNode;



Ponto de Inserção

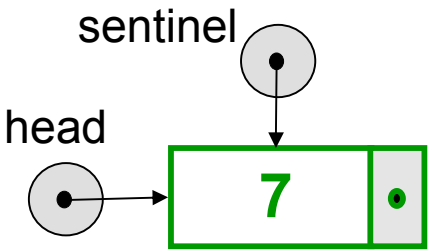
- Note que o fragmento de código encontra o sentinela se a **lista estiver vazia** ou o número inserido tem valor maior do que qualquer outro elemento da lista ordenada
  - sentinel->Entry = x;
  - p = head;
  - while(p->Entry < x)
    - p = p->NextNode;



x | 7

Ponto de Inserção

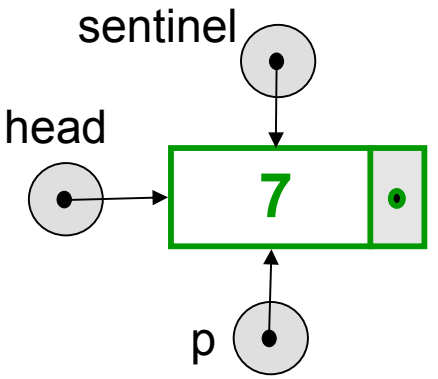
- Note que o fragmento de código encontra o sentinela se a **lista estiver vazia** ou o número inserido tem valor maior do que qualquer outro elemento da lista ordenada
  - **sentinel->Entry = x;**
  - p = head;
  - while(p->Entry < x)
    - p = p->NextNode;



x | 7

Ponto de Inserção

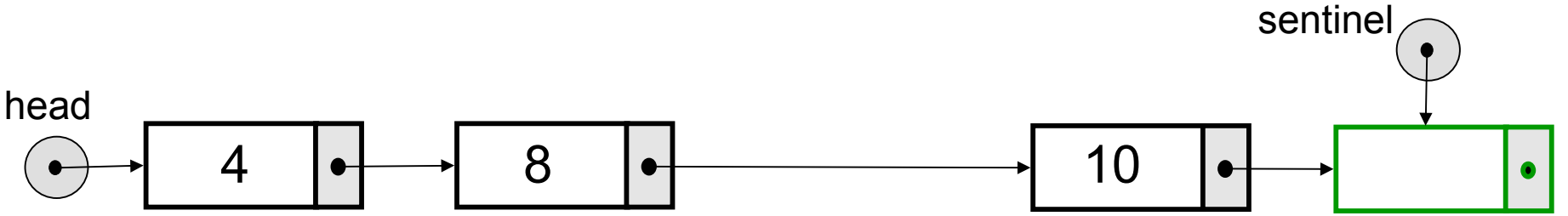
- Note que o fragmento de código encontra o sentinela se a **lista estiver vazia** ou o número inserido tem valor maior do que qualquer outro elemento da lista ordenada
  - sentinel->Entry = x;
  - **p = head;**
  - **while(p->Entry < x)**
    - p = p->NextNode;



x 7

Ponto de Inserção

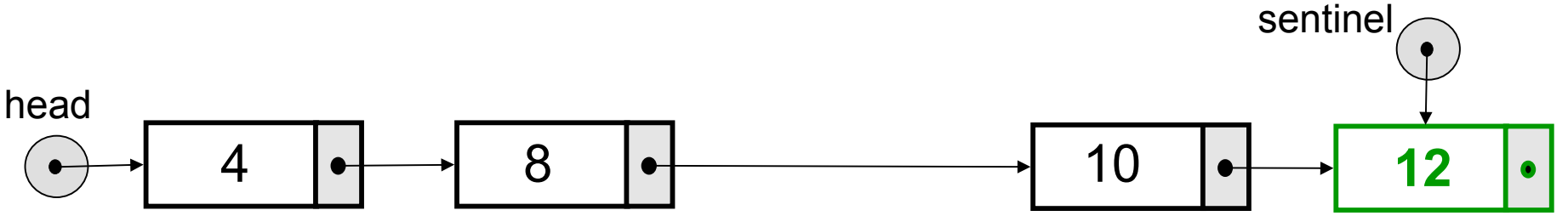
- Note que o fragmento de código encontra o sentinela se a lista estiver vazia ou o **número inserido tem valor maior** do que qualquer outro elemento da lista ordenada
  - sentinel->Entry = x;
  - p = head;
  - while(p->Entry < x)
    - p = p->NextNode;



x 12

Ponto de Inserção

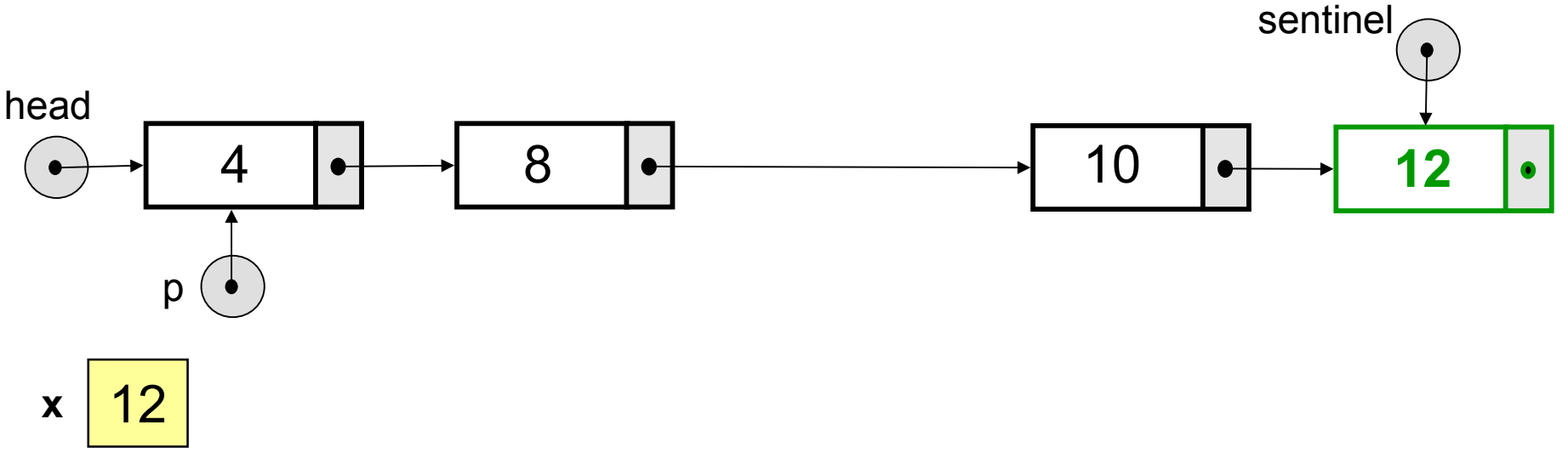
- Note que o fragmento de código encontra o sentinela se a lista estiver vazia ou o **número inserido tem valor maior** do que qualquer outro elemento da lista ordenada
  - **sentinel->Entry = x;**
  - p = head;
  - while(p->Entry < x)
    - p = p->NextNode;



x 12

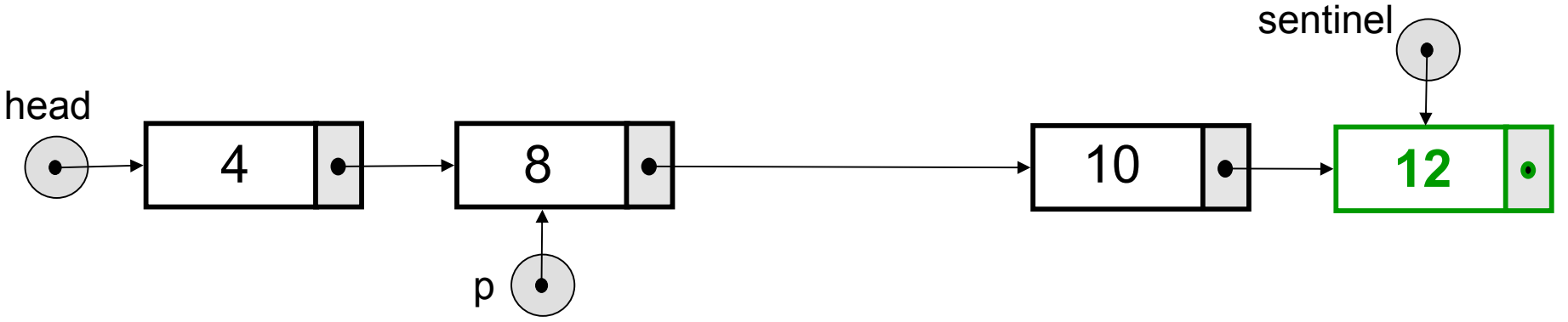
Ponto de Inserção

- Note que o fragmento de código encontra o sentinela se a lista estiver vazia ou o **número inserido tem valor maior** do que qualquer outro elemento da lista ordenada
  - sentinel->Entry = x;
  - **p = head;**
  - **while(p->Entry < x)**
    - p = p->NextNode;



Ponto de Inserção

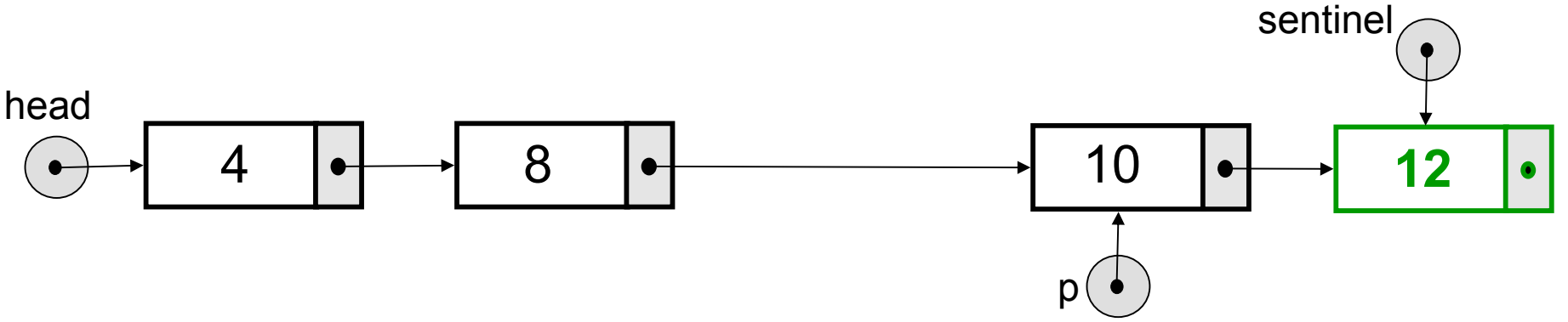
- Note que o fragmento de código encontra o sentinela se a lista estiver vazia ou o **número inserido tem valor maior** do que qualquer outro elemento da lista ordenada
  - sentinel->Entry = x;
  - p = head;
  - **while(p->Entry < x)**
    - **p = p->NextNode;**



x 12

Ponto de Inserção

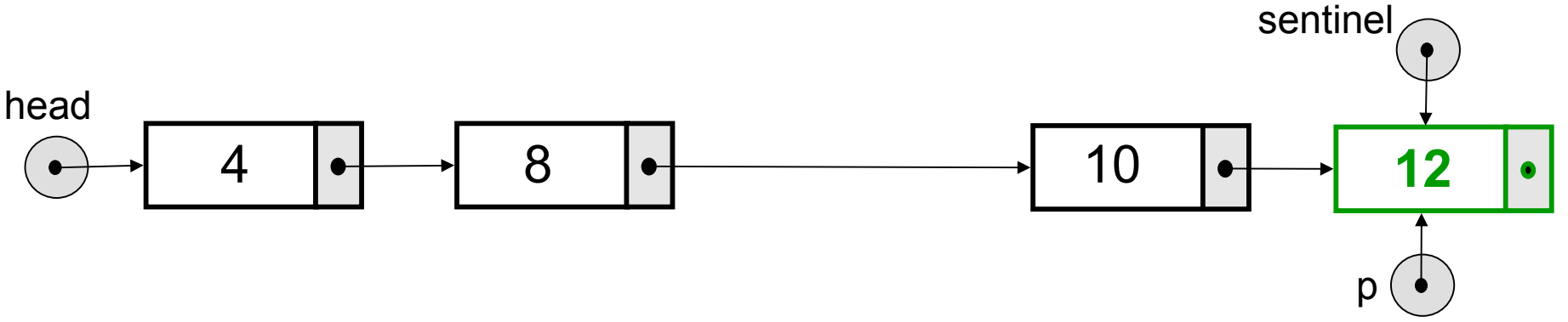
- Note que o fragmento de código encontra o sentinela se a lista estiver vazia ou o **número inserido tem valor maior** do que qualquer outro elemento da lista ordenada
  - sentinel->Entry = x;
  - p = head;
  - **while(p->Entry < x)**
    - **p = p->NextNode;**



x 12

Ponto de Inserção

- Note que o fragmento de código encontra o sentinela se a lista estiver vazia ou o **número inserido tem valor maior** do que qualquer outro elemento da lista ordenada
  - sentinel->Entry = x;
  - p = head;
  - while(p->Entry < x)
    - p = p->NextNode;



x 12

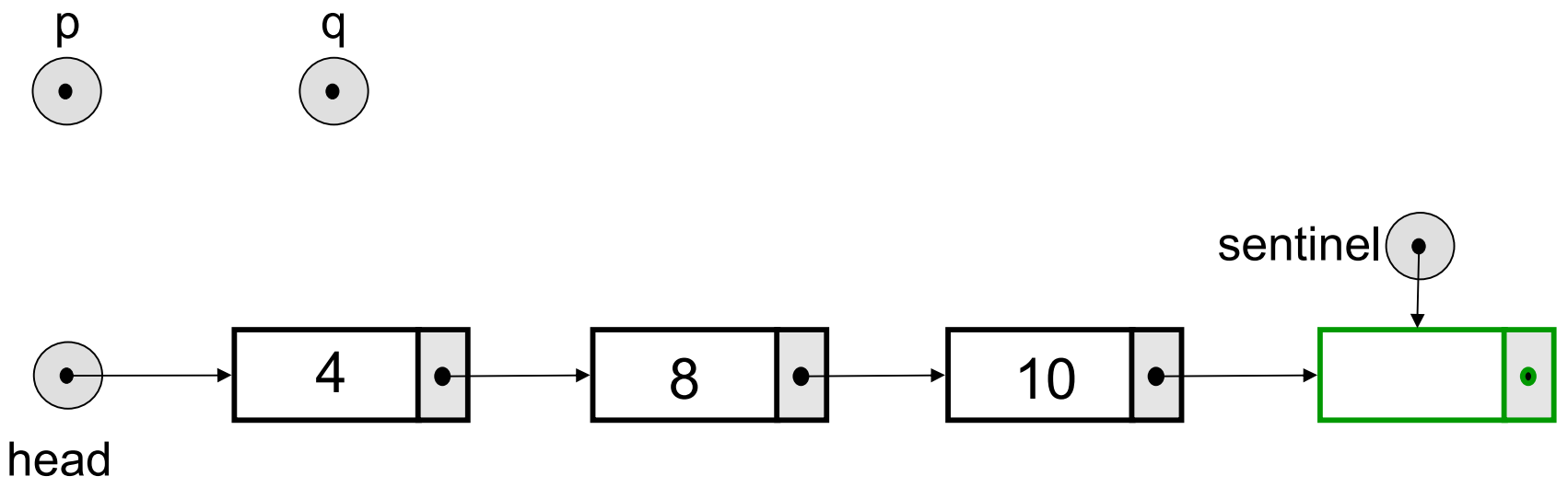
Operações Básicas: Insert

```
void OrderedList::Insert(int x)
{ ListPointer p, q;

  // Buscar local de inserção
  sentinel->Entry = x;
  p = head;
  while(p->Entry < x)
      p = p->NextNode;

  q = new ListNode;
  if(q == NULL)
  { cout << “Memória insuficiente”;
    abort();
  }
  if(p == sentinel)
  {  p->NextNode = q;
     sentinel = q;
  }
  else
  {  *q = *p;
     p->Entry = x;
     p->NextNode = q;
  }
  count++;
}
```

Operações Básicas: Delete

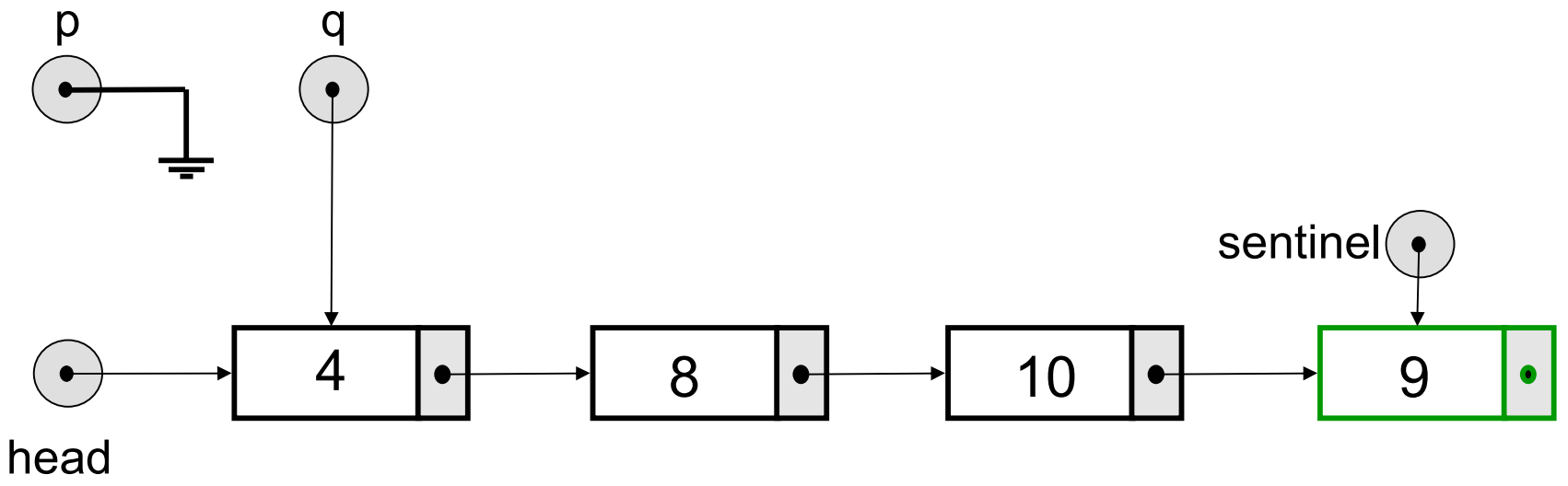
- Diferentemente da estratégia adotada na inserção, na qual apenas um ponteiro percorre a lista ordenada até encontrar o ponto de inserção, a operação de remoção requer o percurso na lista ordenada com dois ponteiros
- Seja **p** um ponteiro que fica sempre uma posição atrás do ponteiro **q** na lista ordenada
- Após encontrar os dois ponteiros **p** e **q** é trivial remover um elemento (ou mesmo inserir, deixado como exercício)

Exemplo: Dois Ponteiros e Busca do Elemento x=9



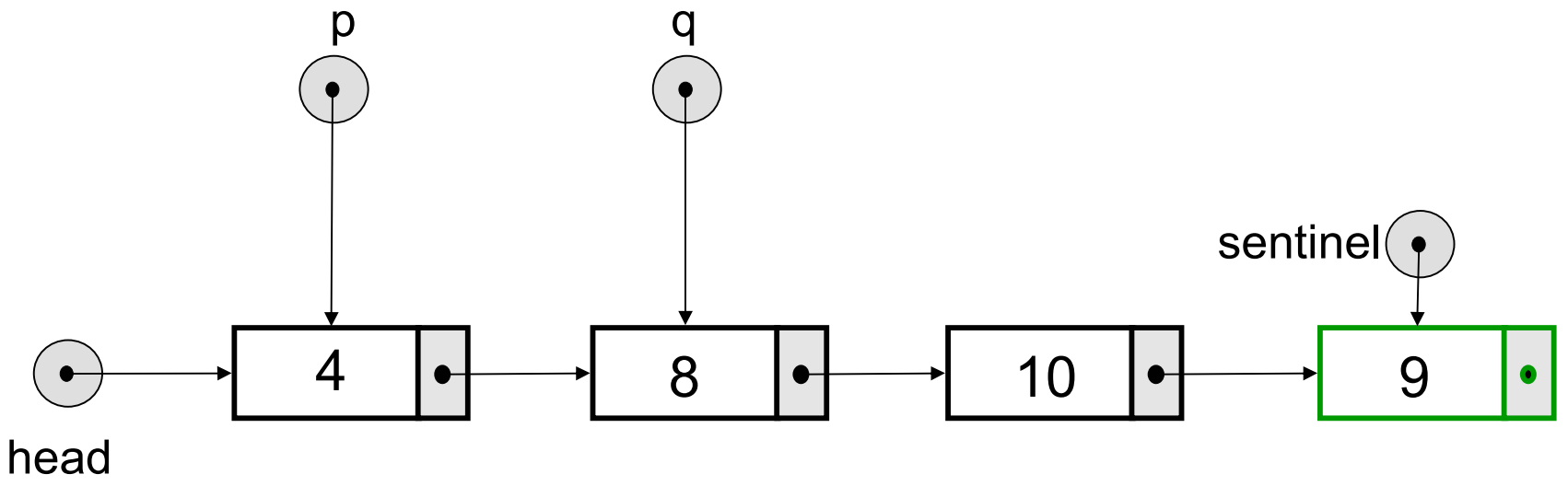
O ponteiro **p** fica sempre uma posição atrás de **q**

Exemplo: Dois Ponteiros e Busca do Elemento x=9



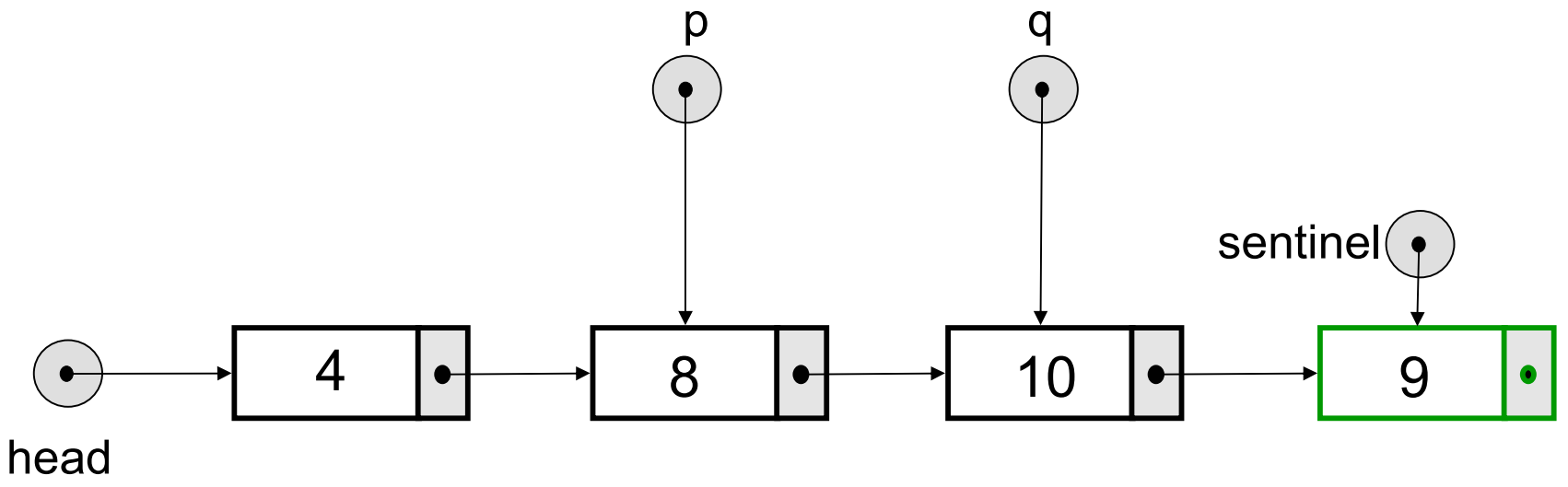
O ponteiro **p** fica sempre uma posição atrás de **q**

Exemplo: Dois Ponteiros e Busca do Elemento x=9



O ponteiro **p** fica sempre uma posição atrás de **q**

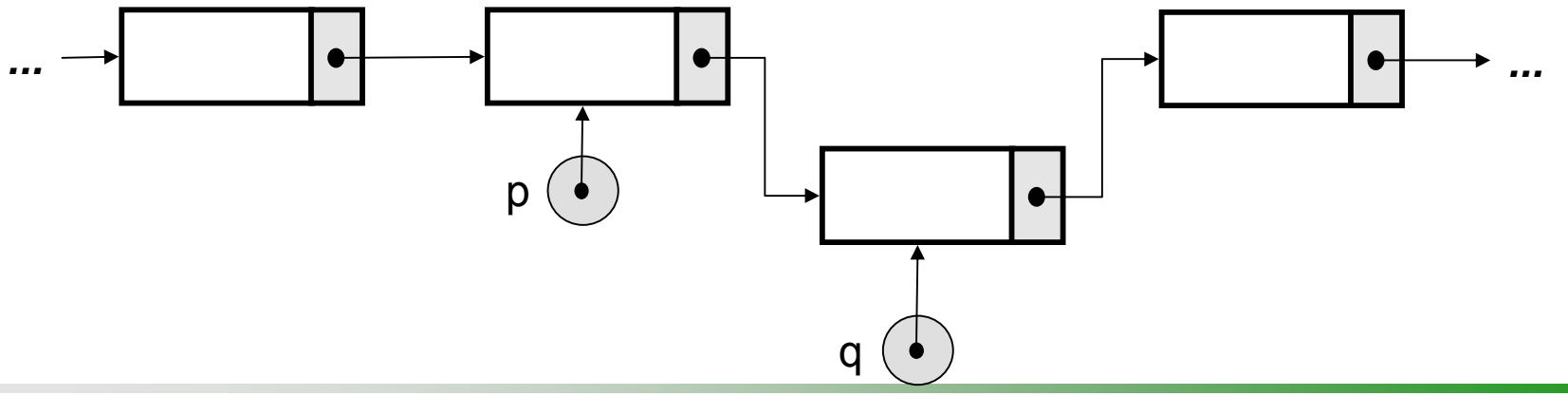
Exemplo: Dois Ponteiros e Busca do Elemento x=9



O ponteiro **p** fica sempre uma posição atrás de **q**

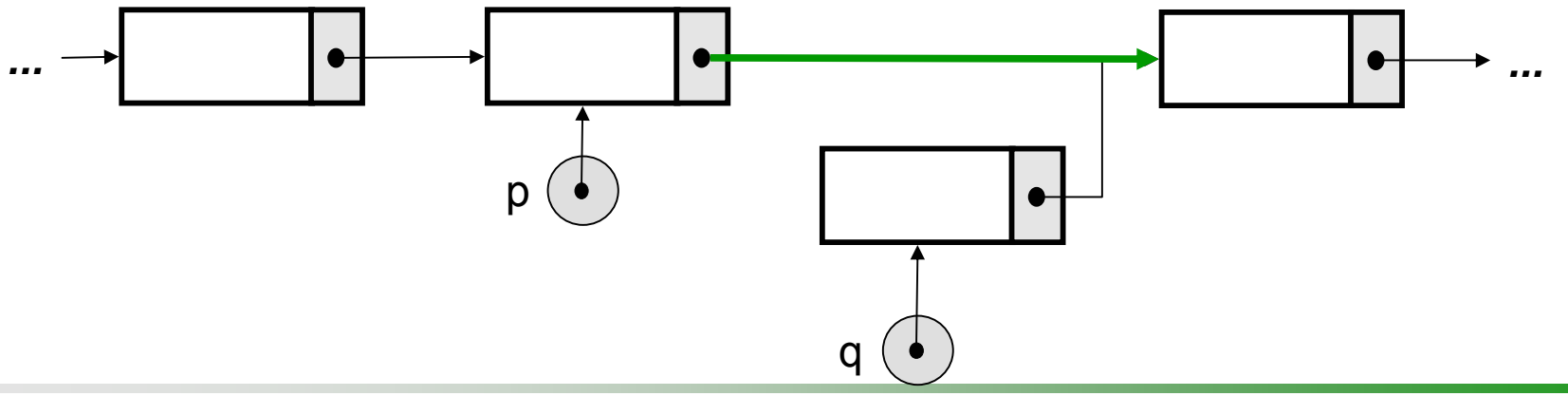
Operações Básicas: Delete

- Uma vez encontrados o ponteiro **q** (elemento a ser removido) e **p** o ponteiro para o nó precedente



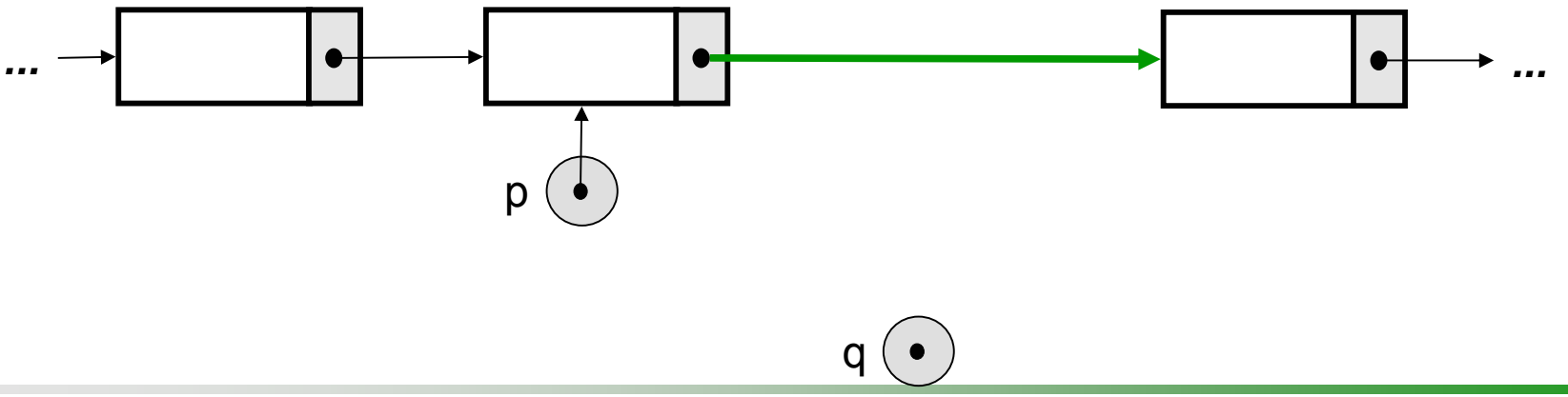
Operações Básicas: Delete

- Uma vez encontrados o ponteiro **q** (elemento a ser removido) e **p** o ponteiro para o nó precedente
  - p->NextNode = q->NextNode;
  - ...



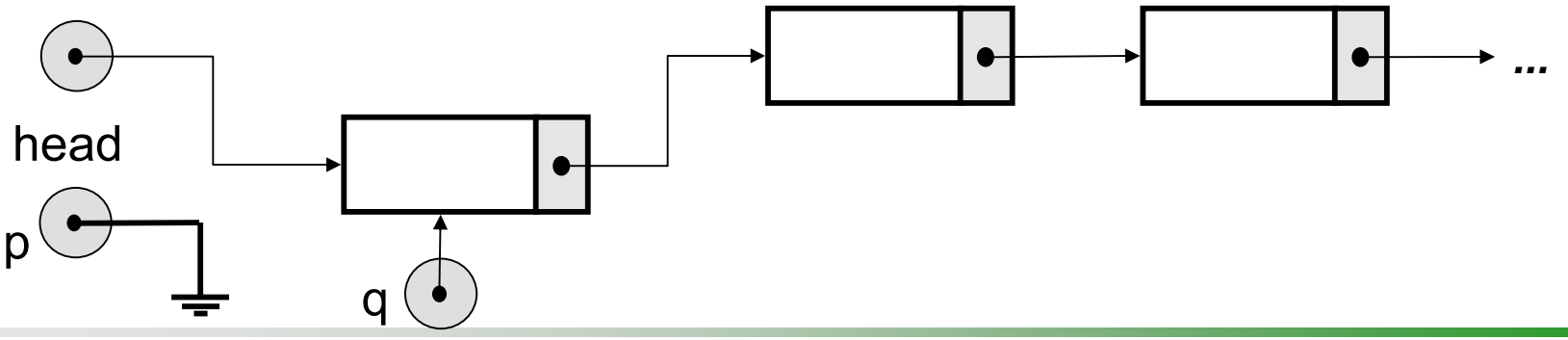
Operações Básicas: Delete

- Uma vez encontrados o ponteiro **q** (elemento a ser removido) e **p** o ponteiro para o nó precedente
  - p->NextNode = q->NextNode;
  - delete q;



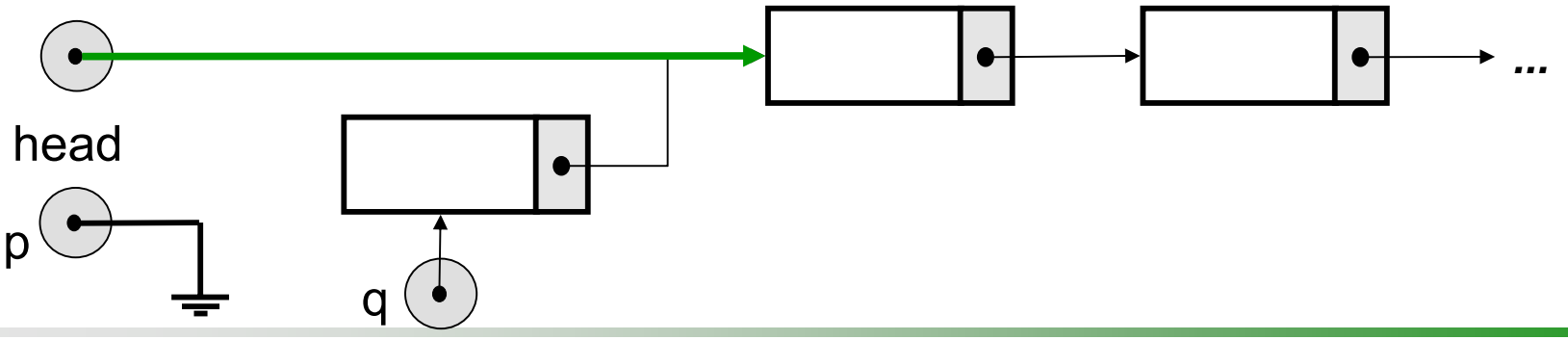
Operações Básicas: Delete

- Entretanto, se a remoção ocorrer no início da lista



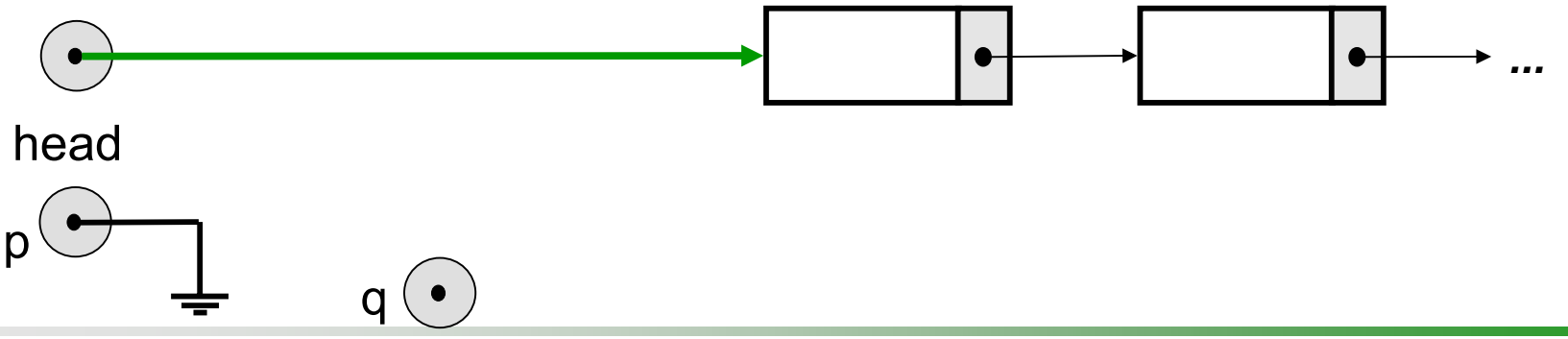
Operações Básicas: Delete

- Entretanto, se a remoção ocorrer no início da lista
  - head = q->NextNode;



Operações Básicas: Delete

- Entretanto, se a remoção ocorrer no início da lista
  - head = q->NextNode;
  - delete q;



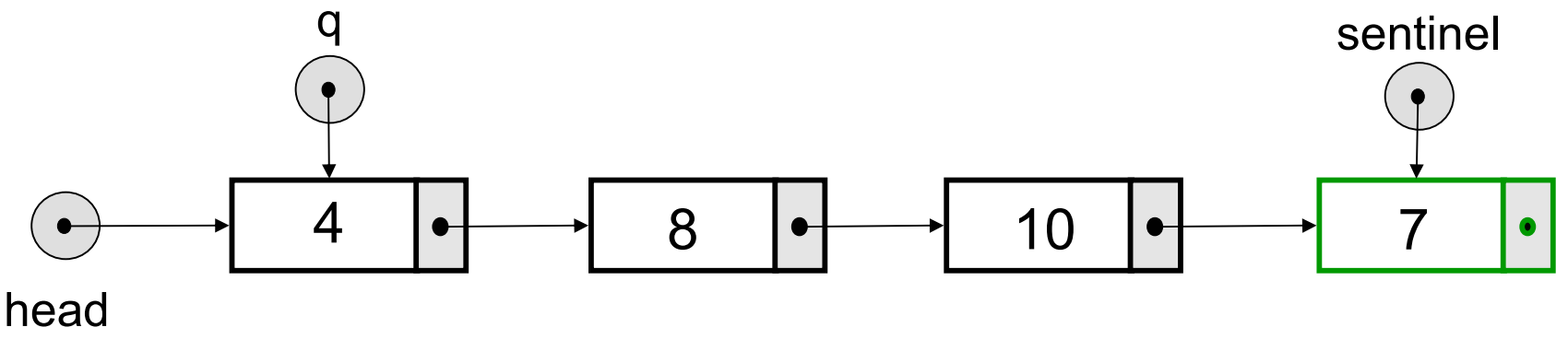
Operações Básicas: Delete

```
void OrderedList::Delete(int x)
{ ListPointer p=NULL, q=head;

  // Buscar local de remoção
  sentinel->Entry = x;
  while(q->Entry < x)
   {  p = q;
      q = q->NextNode;
   }

  // Encontrou x?
 if (q->Entry != x || q == sentinel)
  { cout << “Elemento inexistente”;
    abort();
  }

  // Local de remoção
  if(q == head)
    head = q->NextNode;
  else
    p->NextNode = q->NextNode;

  delete q;
  count = count - 1;
}
```

Operações Básicas: Search

- A operação de busca é simples, considerando o uso de sentinela, como no caso da inserção
- O elemento **x** a ser pesquisado deve ser colocado no campo de dados do sentinela
- A busca do elemento **x** dá-se do início da lista ordenada e prossegue até que um elemento maior ou igual a **x** seja encontrado
- Há três situações possíveis
  - (i) Um elemento maior do que **x** foi encontrado, mas não igual a **x**
  - (ii) Elemento **x** foi encontrado como sendo sentinela
  - (iii) Elemento **x** foi encontrado como parte da lista
- Somente no último caso considera-se que o elemento de pesquisa **x** foi encontrado na lista ordenada

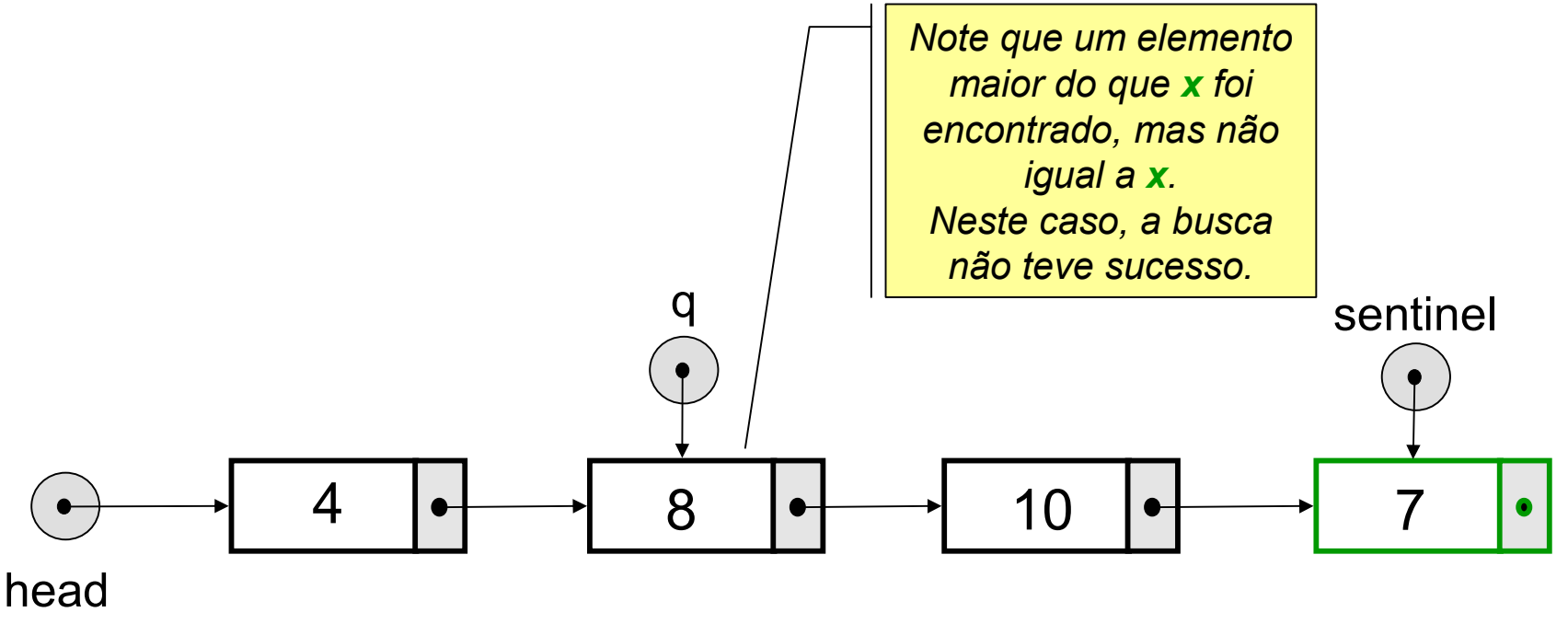
Operações Básicas: Search
(i) Um elemento maior do que **x** foi encontrado, mas não igual a **x**



x 7

A busca continua enquanto o elemento atual < **x**

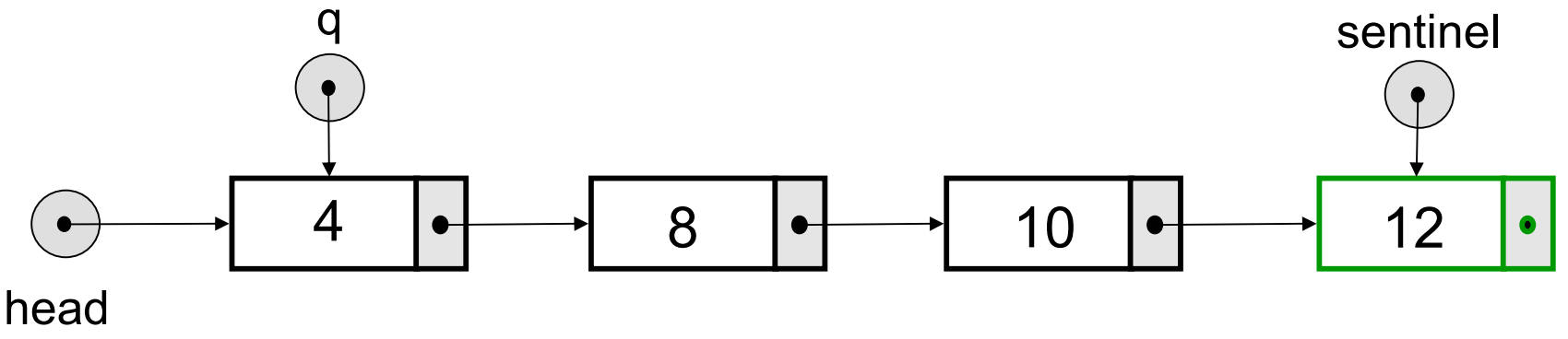
Operações Básicas: Search
(i) Um elemento maior do que **x** foi encontrado, mas não igual a **x**



x | 7

A busca continua enquanto o elemento atual < **x**

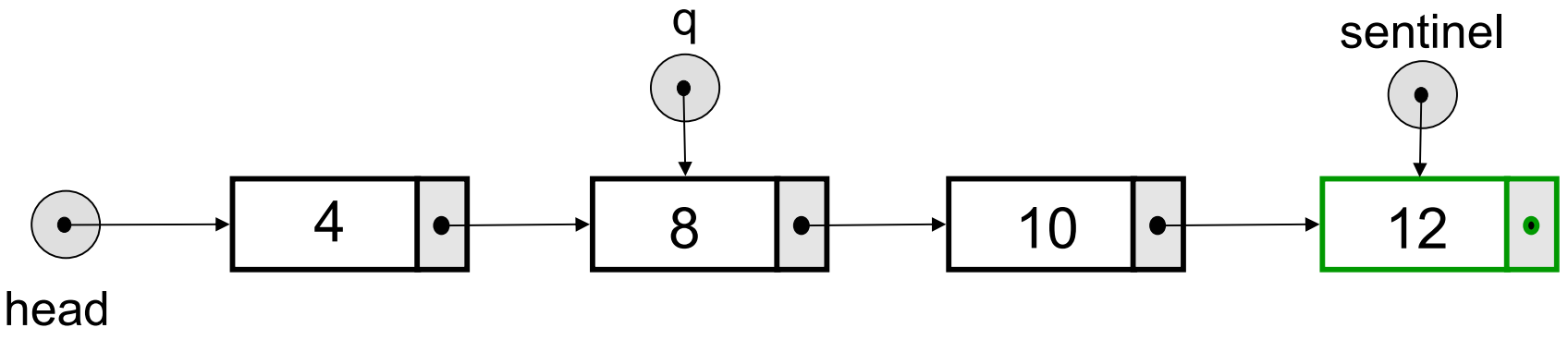
Operações Básicas: Search
(ii) Elemento **x** encontrado como sentinela



x 12

A busca continua enquanto o elemento atual < **x**

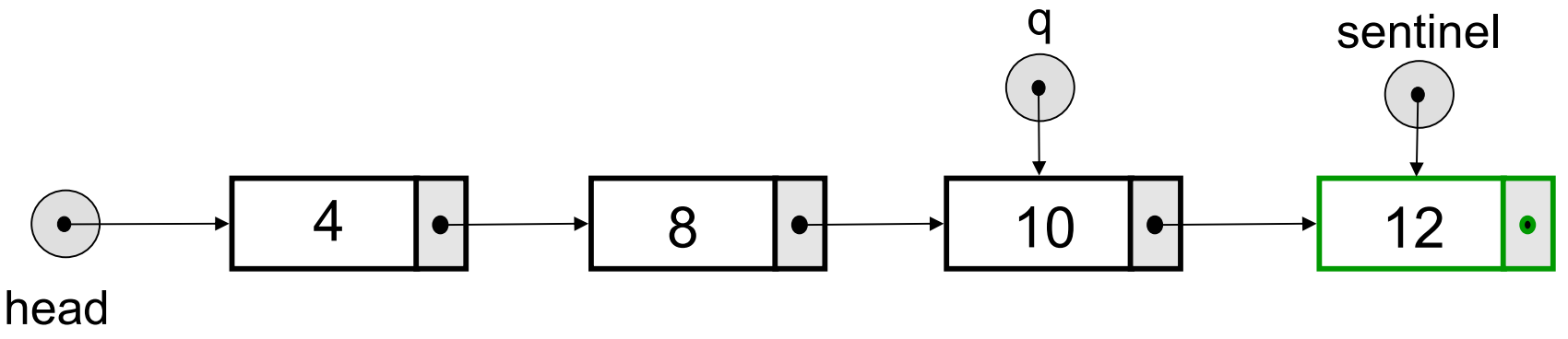
Operações Básicas: Search
(ii) Elemento **x** encontrado como sentinela



x 12

A busca continua enquanto o elemento atual < **x**

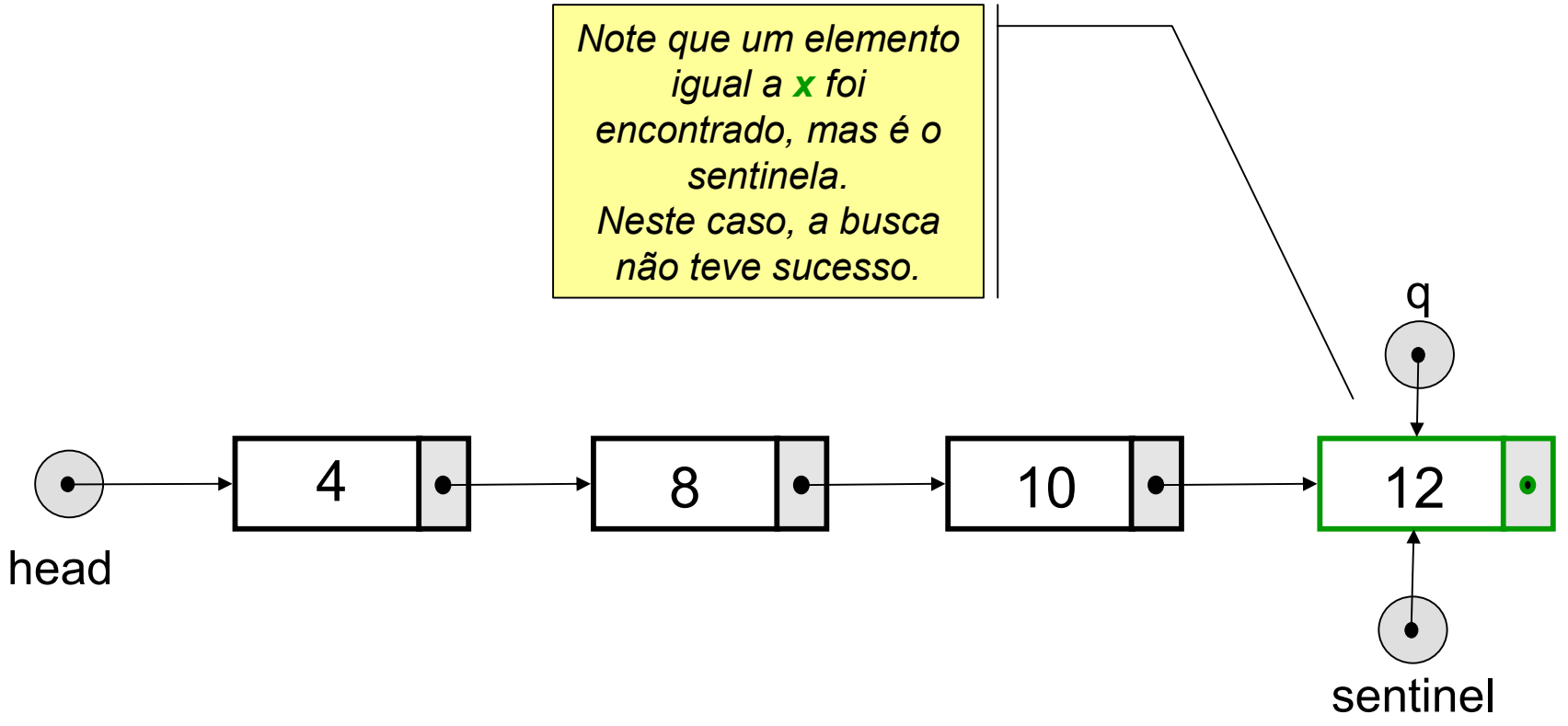
Operações Básicas: Search
(ii) Elemento **x** encontrado como sentinela



x 12

A busca continua enquanto o elemento atual < **x**

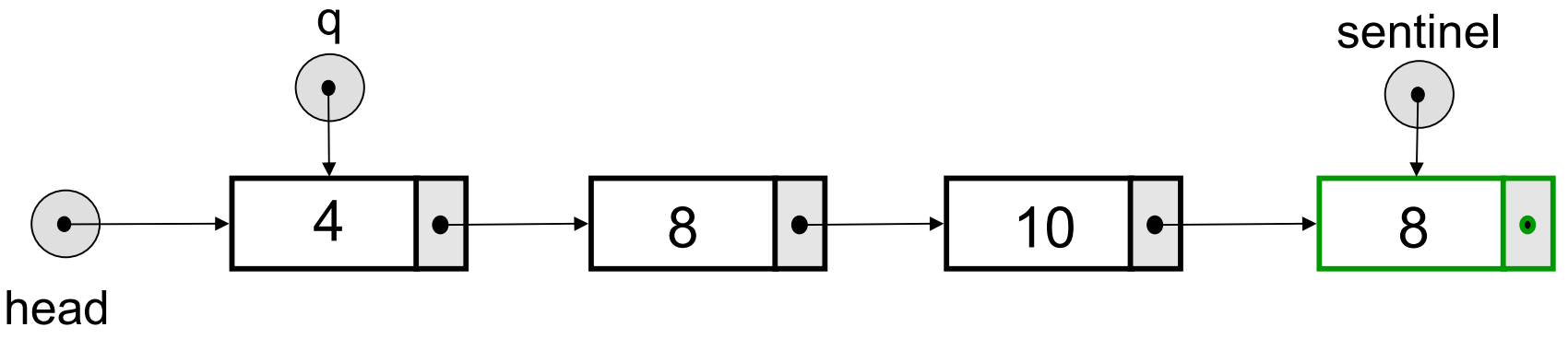
Operações Básicas: Search
(ii) Elemento **x** encontrado como sentinela



x 12

A busca continua enquanto o elemento atual < **x**

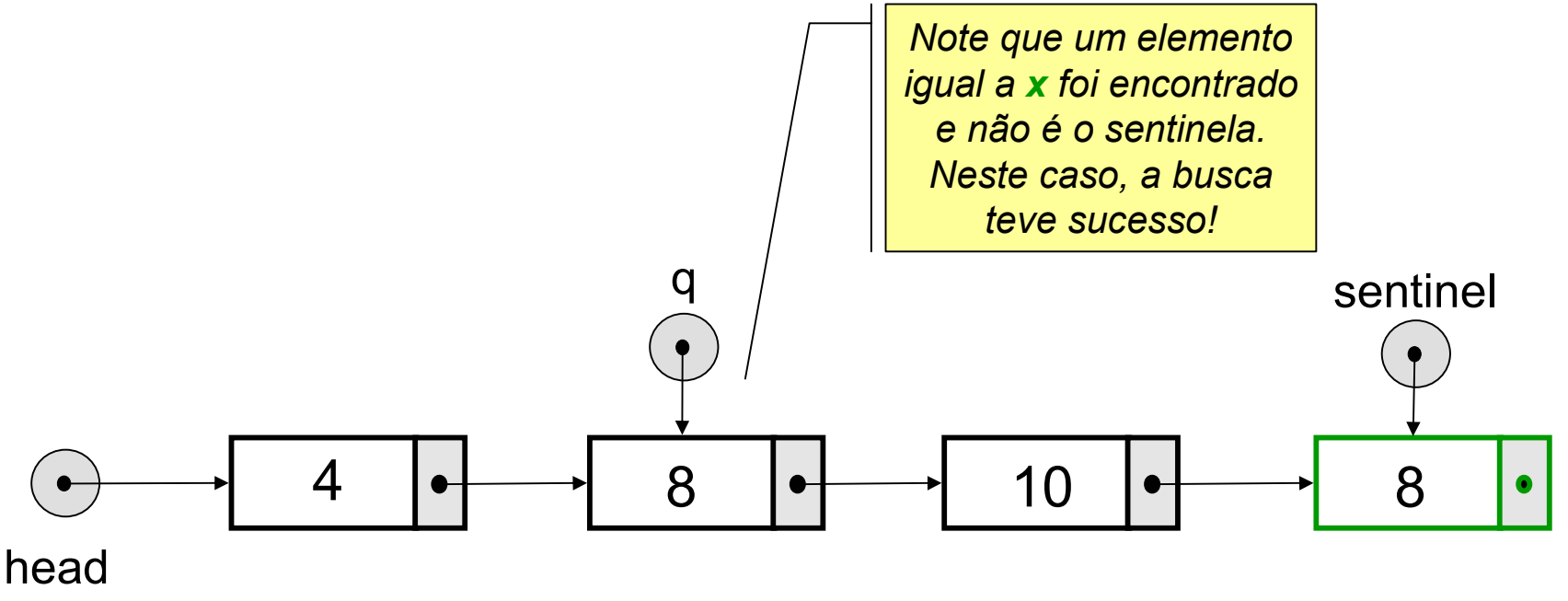
Operações Básicas: Search
(iii) Elemento **x** encontrado como parte da lista



x 8

A busca continua enquanto o elemento atual < **x**

Operações Básicas: Search
(iii) Elemento **x** encontrado como parte da lista



x 8

A busca continua enquanto o elemento atual < **x**

Operações Básicas: Search

```
int OrderedList::Search(int x)
{ int posicao=1;
  ListPointer q=head;

  sentinel->Entry = x;
  while (q->Entry < x)
  {  q = q->NextNode;
     posicao++;
  }

  if (q->Entry != x || q == sentinel)
    return 0;
  else
    return posicao;
}
```

Busca com Inserção

- Em muitos problemas é interessante considerar a situação em que se um elemento não se encontra na lista então ele deve ser inserido; caso contrário, um contador deve ser atualizado
- Um exemplo típico é a contagem do número de palavras em um texto
- Para tanto, nossa estrutura de dados será estendida para a seguinte definição

```
struct ListNode
{  string Entry;          // chave
   int count;             // contador
   ListNode *NextNode;
};
```

Busca com Inserção

```
void OrderedList::SearchInsert(int x)
{ ListPointer p, q;

  // Buscar elemento ou local de inserção
  sentinel->Entry = x;
  p = head;
  while(p->Entry < x)
      p = p->NextNode;

 if(p != sentinel && p->Entry == x)
    p->count++; // encontrou
 else
 {  // não encontrou, inserir
    q = new ListNode;
    if(q == NULL)
    { cout << “Memória insuficiente”;
      abort();
    }
    if(p == sentinel)
    {   p->NextNode = q;
        sentinel = q;
     }
     else
     {  *q = *p;
        p->Entry = x;
        p->NextNode = q;
     }
     p->count = 1;
     count++;
  }
}
```

Exercícios

- Implemente as demais operações em listas
  - Clear()
  - Size()

Solução Clear/Size

- Note que o código é diferente do destruidor, já que o sentinela deve permanecer

```
void OrderedList::Clear()
{ ListPointer q;

  while (head != sentinel)
   {  q = head;
      head = head->NextNode;
      delete q;
   }
   count = 0;
}
```

```
int OrderedList::Size()
{
   return count;
}
```

Considerações Finais

- A maior parte das operações em listas encadeadas de tamanho **n** (ordenadas ou não) tem tempo O(1)
  - Busca, inserção ou remoção têm tempo O(**n**) com ou sem uso de sentinelas
- Veremos nas próximas aulas estruturas de dados não lineares, as árvores, que possuem características que permitem melhorar os tempos de busca, inserção ou remoção