



# Listas Lineares



Algoritmos e  
Estruturas de Dados I

- ❑ Nas aulas anteriores vimos pilhas e filas que são listas restritas
- ❑ Nesta aula veremos o ADT **lista** em sua forma genérica, também conhecido com **lista linear**

# Introdução

---

- ❑ Quando estudamos pilhas, praticamos *information hiding* — ocultamento da informação — ao separar o uso de pilhas da sua forma real de programação (implementação) de suas operações
- ❑ Ao estudar Listas, continuamos com este método e pudemos perceber que algumas variações na implementação são possíveis
- ❑ Com listas genéricas, nós temos muito mais flexibilidade e liberdade no acesso e alteração das entradas (itens ou nós) em qualquer parte da lista
- ❑ Os princípios de *information hiding* são ainda mais importantes para listas genéricas do que para listas restritas (pilhas e filas)

# Organização

---

- ❑ Definição do ADT Lista

- ❑ Especificação

- Operações sobre o ADT Lista, utilizando pré- e pós-condições

- ❑ Implementação

- Estática (contígua)
- Dinâmica (encadeada)

# Definição

---

- ❑ Um dos objetos de dados mais simples e utilizado é a **lista linear**
- ❑ Um exemplo de lista são os dias da semana: [segunda, terça, quarta, quinta, sexta, sábado, domingo]
- ❑ Outro exemplo são os valores de um baralho de cartas: [2, 3, 4, 5, 6, 7, 8, 9, 10, Valete, Dama, Reis, Ás]
- ❑ Ou andares de um edifício: [subsolo, térreo, sobreloja, primeiro, segundo, ...]

# Definição

---

- Uma **lista linear** é uma seqüência de **n** elementos  **$[a_1, a_2, \dots, a_i, \dots, a_n]$** 
  - 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  **$a_n$**
- Em uma lista elementos podem ser **inseridos**, **removidos** ou **substituídos** em qualquer posição

# Definição

---

- Ao inserir um elemento  $x$  na posição  $p$  ( $1 \leq p \leq n+1$ ), os elementos  $a_i$  são **deslocados** para as posições  $a_{i+1}$ , ( $p \leq i \leq n$ )
  - Ou seja, os elementos  $a_p, a_{p+1}, \dots, a_n$  são deslocados para as posições  $a_{p+1}, \dots, a_n, a_{n+1}$  e o elemento  $x$  é inserido na posição  $a_p$
- Ao remover um elemento  $x$  da posição  $p$  ( $1 \leq p \leq n$ ), os elementos  $a_i$  são **deslocados** para as posições  $a_{i-1}$ , ( $p \leq i \leq n$ )
  - Ou seja, o elemento  $x$  é removido da posição  $a_p$  e os elementos  $a_{p+1}, \dots, a_n$  são deslocados para as posições  $a_p, \dots, a_{n-1}$

# Exemplo

---

- Seja a lista  $L = [1, 7, 2]$  contendo  $n=3$  elementos
  - $a_1=1; a_2=7, a_3=2$
- Se inserimos o elemento **9** na 2ª posição da lista, teremos  $L = [1, 9, 7, 2]$  com  $n=4$  elementos
  - $a_1=1; a_2=9, a_3=7, a_4=2$
  - Observe que os elementos **7** e **2** foram deslocados em uma posição na lista
- Se após isso inserimos o elemento **3** na 5ª posição da lista, teremos  $L = [1, 9, 7, 2, 3]$  com  $n=5$  elementos
  - $a_1=1; a_2=9, a_3=7, a_4=2, a_5=3$
  - Neste caso, o elemento **3** foi inserido no final da lista

# Exemplo

---

- Ainda considerando  $L = [1, 9, 7, 2, 3]$  com  $n=5$  elementos
  - $a_1=1; a_2=9, a_3=7, a_4=2, a_5=3$
- Se removermos o primeiro elemento, teremos  $L=[9, 7, 2, 3]$  com  $n=4$  elementos
  - $a_1=9; a_2=7, a_3=2, a_4=3$
  - Observe que os elementos **9**, **7**, **2** e **3** foram deslocados em uma posição na lista
- Se após isso removermos o segundo elemento, teremos  $L = [9, 2, 3]$  com  $n=3$  elementos
  - $a_1=9; a_2=2, a_3=3$
  - Observe que os elementos **2** e **3** foram deslocados em uma posição na lista

# Especificação

---

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

# Criação

---

List::List();

□ *pré-condição*: nenhuma

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

# Destruição

---

List::~~List();

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

# Status

---

`bool List::Empty();`

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

# Status

---

bool List::Full();

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

# Operações Básicas

---

void List::Insert(int p, ListEntry x);

- *pré-condição*: Lista já tenha sido criada, não está cheia e  $1 \leq p \leq n+1$ , onde **n** é o número de entradas na Lista
- *pós-condição*: O item **x** é armazenado na posição **p** na Lista e todas as entradas seguintes (desde que  $p \leq n$ ) têm suas posições incrementada em uma unidade

# Operações Básicas

---

void List::Insert(int p, ListEntry x);

□ *pré-condição*: Lista já tenha sido criada, não está cheia e  $1 \leq p \leq n+1$ , onde **n** é o número de entradas na Lista

□ *pós-condição*: O item **x** é inserido na posição **p** na Lista e todos os elementos seguintes (desde que existam) são deslocados para posições incrementadas.

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 List::Delete(int p, ListEntry &x);`

- *pré-condição*: Lista já tenha sido criada, não está vazia e  $1 \leq p \leq n$ , onde **n** é o número de entradas na Lista
- *pós-condição*: A entrada da posição **p** é removida da Lista e retornada na variável **x**; as entradas de todas as posições seguintes (desde que  $p < n$ ) têm suas posições decrementadas em uma unidade

# Operações Básicas

---

`void List::Retrieve(int p, ListEntry &x);`

- *pré-condição*: Lista já tenha sido criada, não está vazia e  $1 \leq p \leq n$ , onde **n** é o número de entradas na Lista
- *pós-condição*: A entrada da posição **p** da Lista é retornada na variável **x**; a Lista permanece inalterada

# Operações Básicas

---

void List::Replace(int p, ListEntry x);

- *pré-condição*: Lista já tenha sido criada, não está vazia e  $1 \leq p \leq n$ , onde **n** é o número de entradas na Lista
- *pós-condição*: A entrada da posição **p** da Lista é substituída por **x**; as demais entradas da Lista mantêm-se inalteradas

# Outras Operações

---

`void List::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 List::Size();`

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

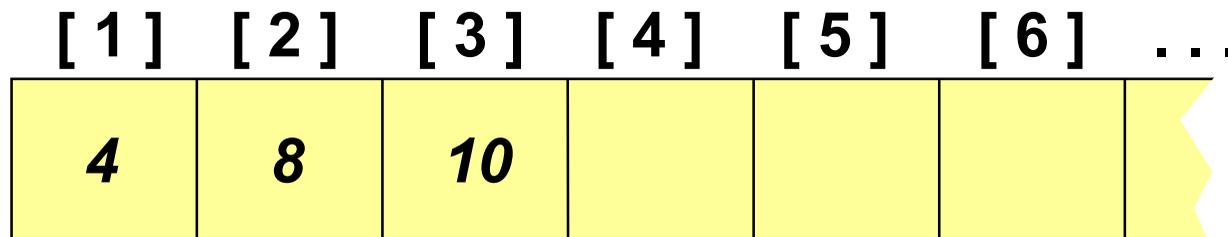
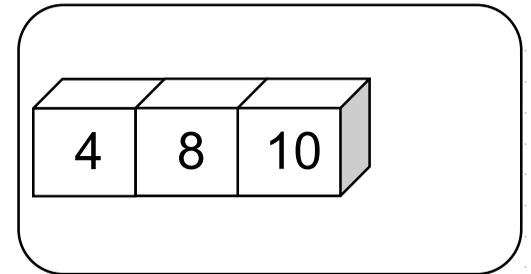
# Implementação Contígua

---

- ❑ A implementação mais simples de uma lista linear consiste em utilizar um vetor no qual associamos o elemento da lista  $a_i$  com o índice do vetor  $i$
- ❑ Essa representação permite recuperar ou modificar os valores dos elementos de uma lista em tempo constante
- ❑ Apenas as operações de inserção e remoção exigem um esforço maior, deslocando alguns elementos restantes para que o mapeamento seqüencial seja preservado em sua forma correta

# Implementação Contígua

- As entradas em uma Lista serão inicialmente armazenadas no início de um vetor, como mostra este exemplo

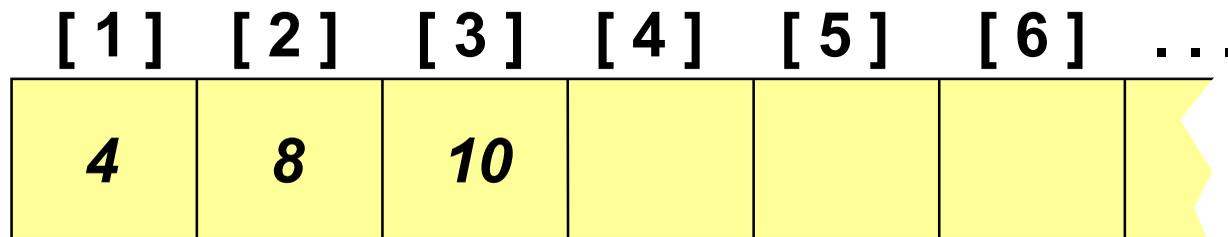
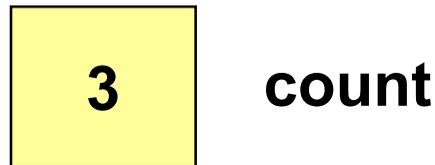
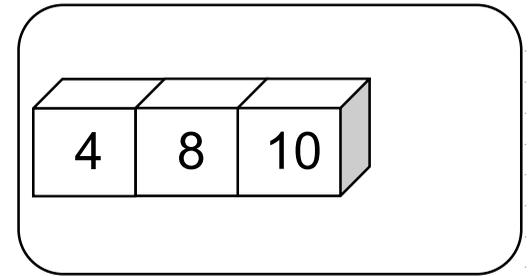


Um vetor de inteiros

Nós não nos interessamos para o que está armazenado nesta parte do vetor

# Implementação Contígua

- Um contador indica a quantidade de elementos na lista



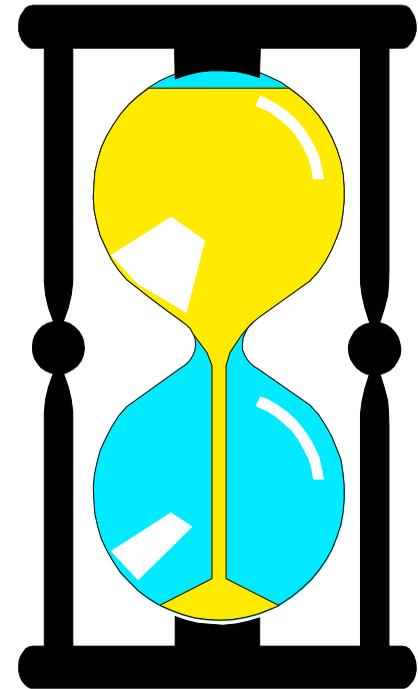
Um vetor de inteiros

Nós não nos interessamos para o que está armazenado nesta parte do vetor

# Questão

---

❑ Utilize estas idéias para escrever uma declaração de tipo que poderia implementar o tipo de dado lista. A declaração deve ser um objeto com dois campos de dados. Faça uma lista capaz de armazenar 100 inteiros



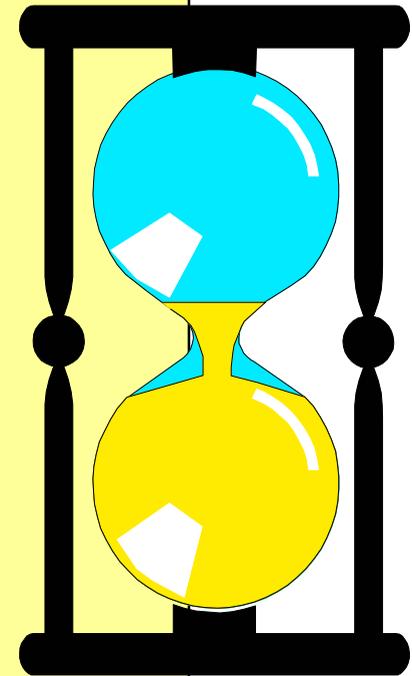
Você tem 3 minutos para escrever a declaração

# Uma Solução

---

```
const int MaxList = 100;
class List
{ public:
    List();           // construtor
    void Insert(int p, int x);
    void Delete(int p, int &x);

    ...
private:
    int count;       // nº. de elementos lista
    int Entry[MaxList+1]; // vetor com elementos
};
```



# Uma Solução

---

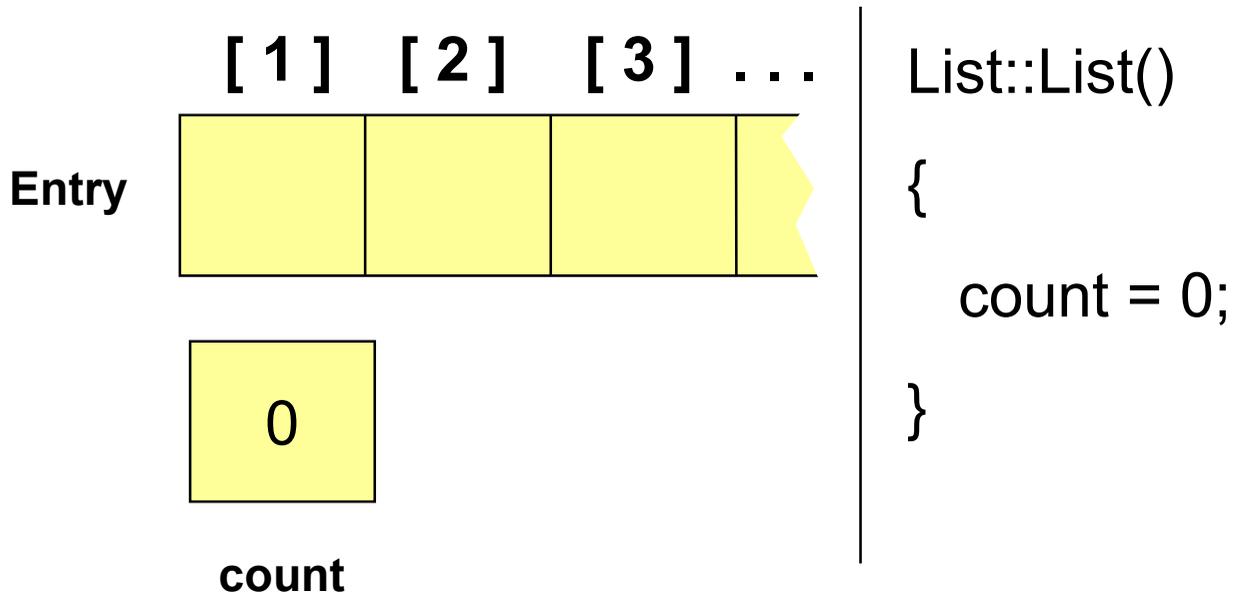
```
const int MaxList = 100;
class List
{ public:
    List();           // construtor
    void Insert(int p, int x);
    void Delete(int p, int &x);
    ...
private:
    int count;       // nº. de elementos lista
    int Entry[MaxList+1]; // vetor com elementos
};
```

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

# Construtor

```
List::List()
```

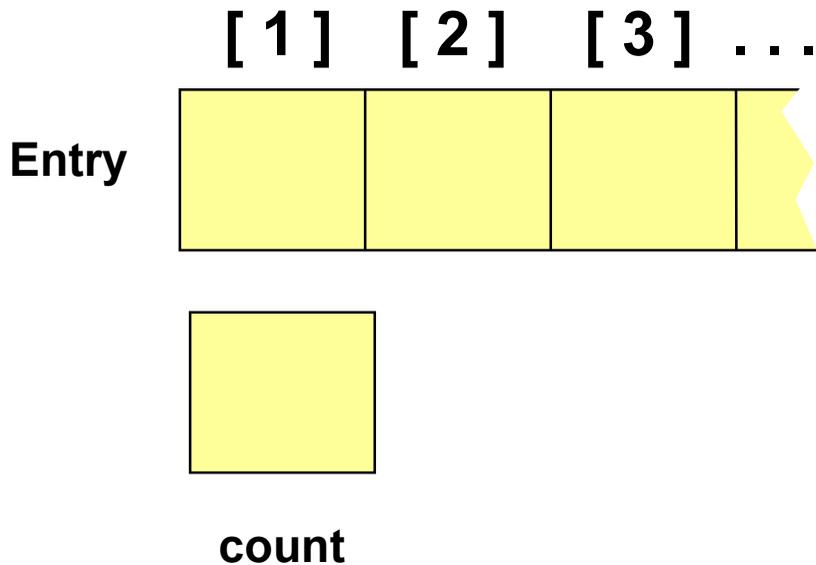
Numa lista vazia não temos nenhum elemento...



# Destruidor

```
List::~~List()
```

Usando alocação estática para implementar a lista, o destruidor não será necessário. Em todo caso, colocaremos apenas uma mensagem que o objeto foi destruído



```
List::~~List()
```

```
{
```

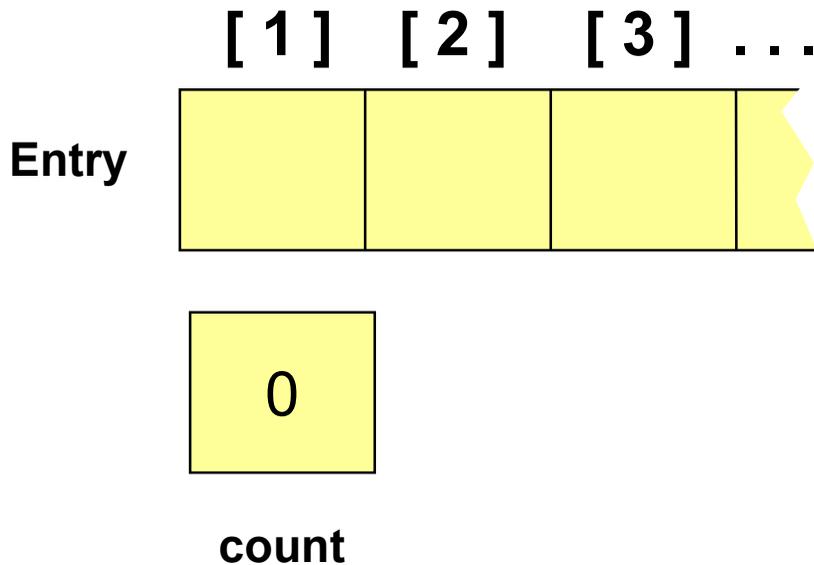
```
    cout << "Lista destruída";
```

```
}
```

# Status: Empty

```
bool List::Empty()
```

Lembre-se que a lista possui um contador de elementos...

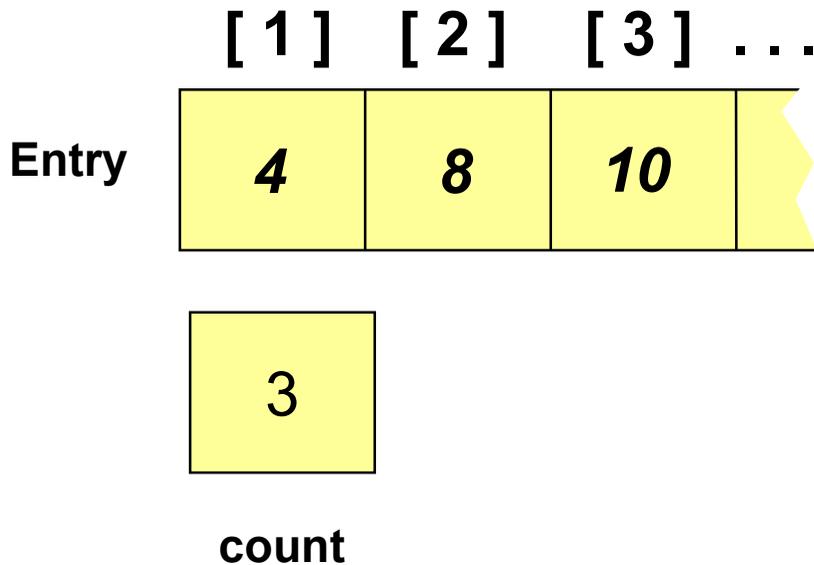


```
bool List::Empty()
{
    return (count == 0);
}
```

# Status: Full

```
bool List::Full()
```

... e que **MaxList** é o número máximo de elementos da lista.

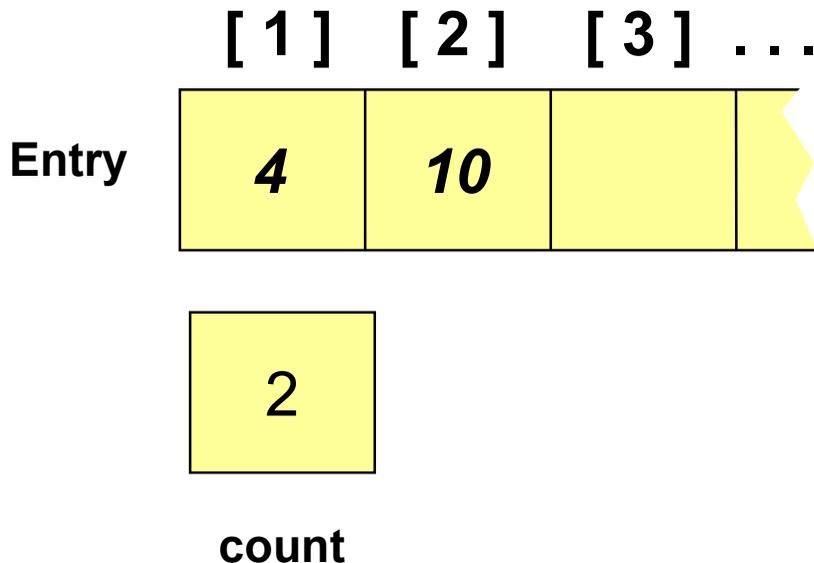


```
bool List::Full()
{
    return (count == MaxList);
}
```

# Operações Básicas: Insert

```
void List::Insert(int p, int x)
```

Nós fazemos uma chamada a  
Insert(2,8)

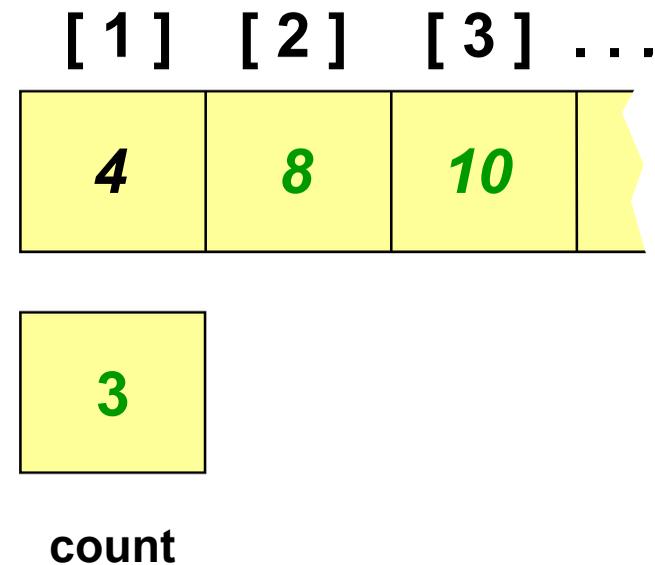
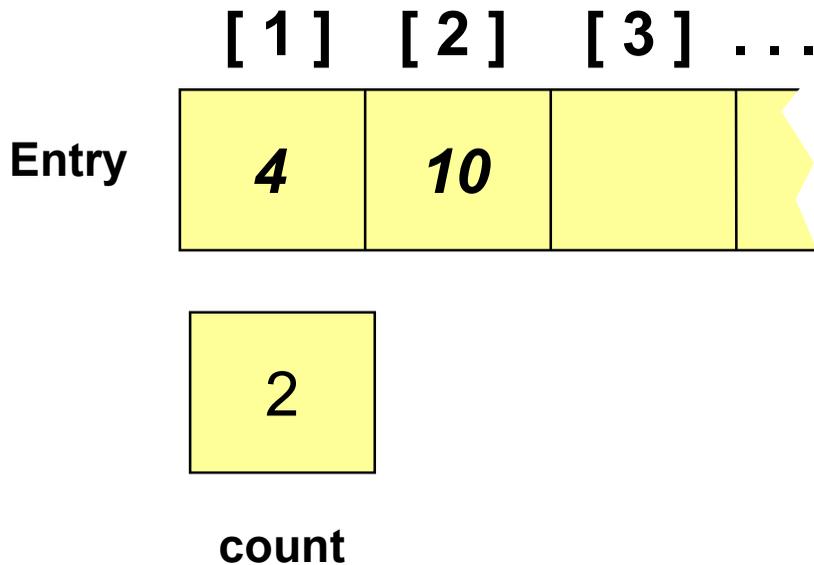


Quais valores serão armazenados em **Entry** e **count** depois que a chamada de procedimento termina?

# Operações Básicas: Insert

```
void List::Insert(int p, int x)
```

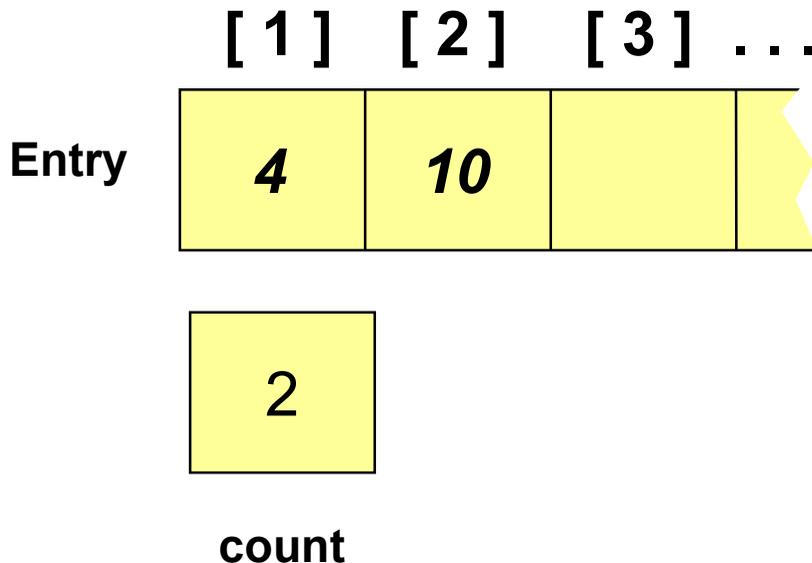
Depois da chamada a `Insert(2,8)`, nós teremos esta lista:



# Operações Básicas: Insert

```
void List::Insert(int p, int x)
```

Antes de inserir, é conveniente verificar se há espaço na lista...

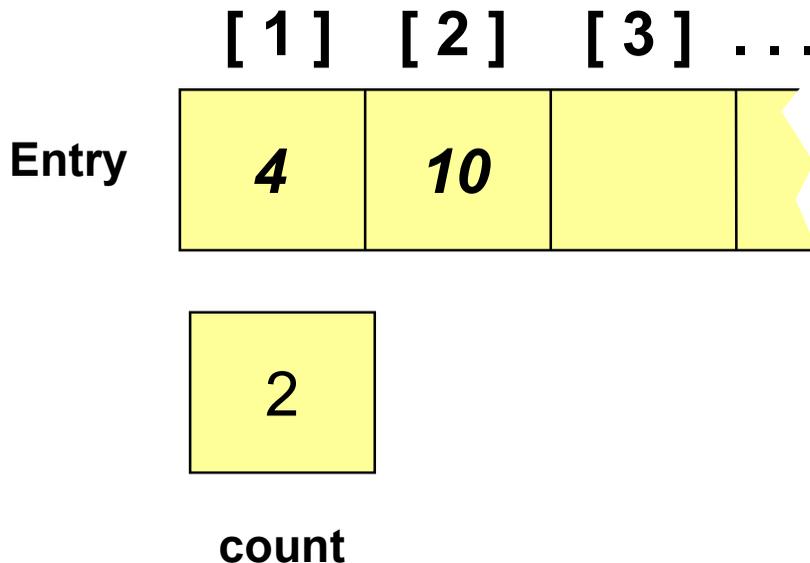


```
void List::Insert(int p,int x)
{ int i;
  if (Full())
  { cout << "Lista Cheia";
    abort();
  }
  ...
}
```

# Operações Básicas: Insert

```
void List::Insert(int p, int x)
```

...e se a posição de inserção fornecida é válida

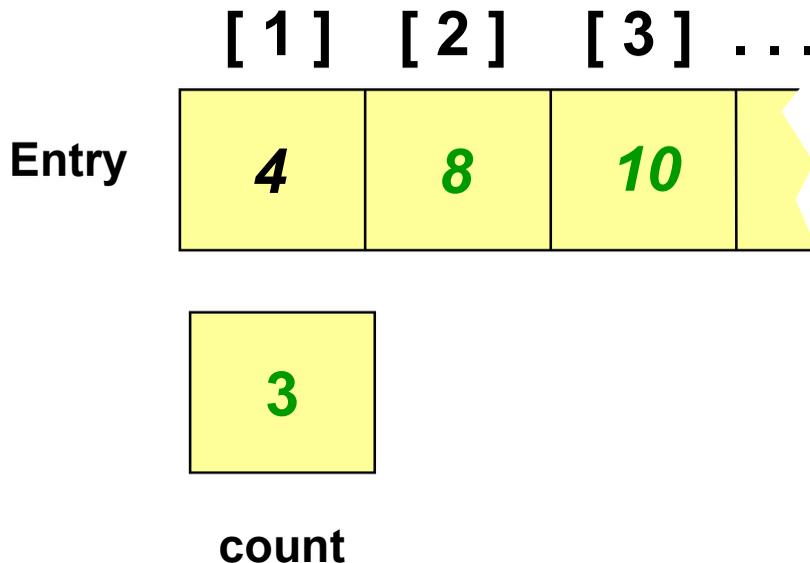


```
void List::Insert(int p,int x)
{ int i;
  if (Full())
  { cout << "Lista Cheia";
    abort();
  }
  if (p < 1 || p > count+1)
  { cout << "Posição inválida";
    abort();
  }
  ...
}
```

# Operações Básicas: Insert

```
void List::Insert(int p, int x)
```

Agora basta deslocar os elementos e inserir **x** na posição **p**

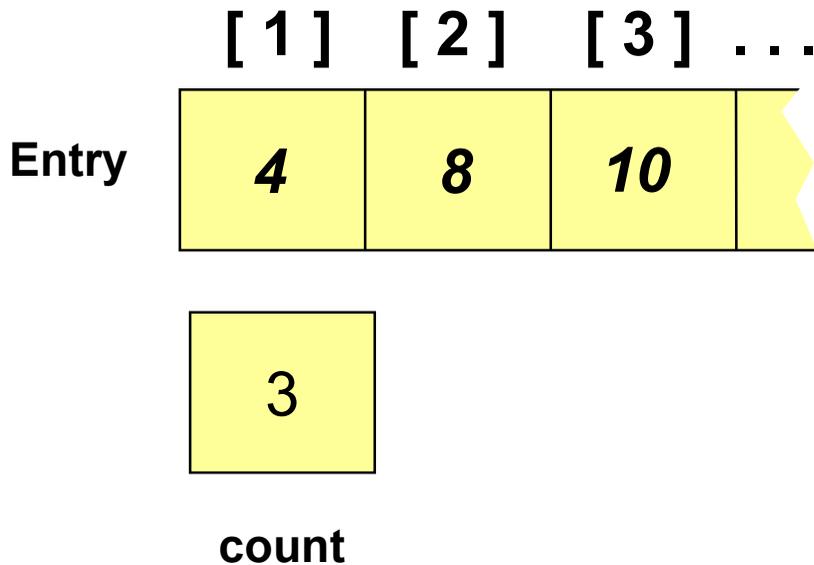


```
void List::Insert(int p,int x)
{ int i;
  if (Full())
  { cout << "Lista Cheia";
    abort();
  }
  if (p < 1 || p > count+1)
  { cout << "Posição inválida";
    abort();
  }
  for(i=count; i>=p; i - -)
    Entry[i+1] = Entry[i];
  Entry[p] = x;
  count++;
}
```

# Operações Básicas: Delete

```
void List::Delete(int p, int &x)
```

Nós fazemos uma chamada a  
Delete(1,x)

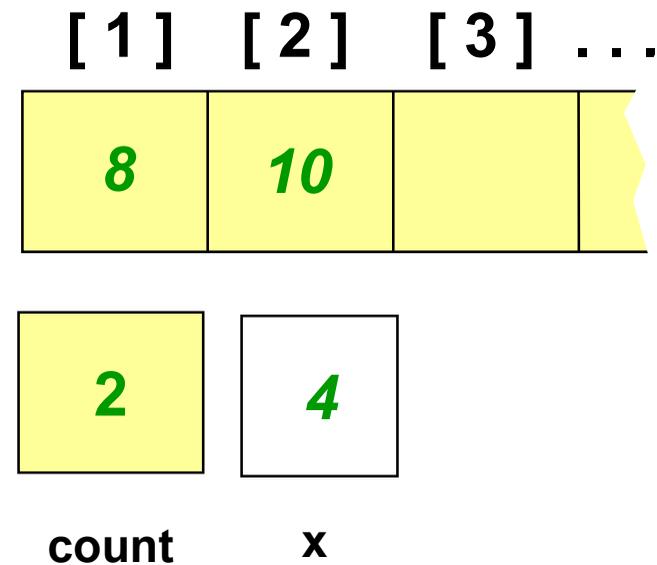
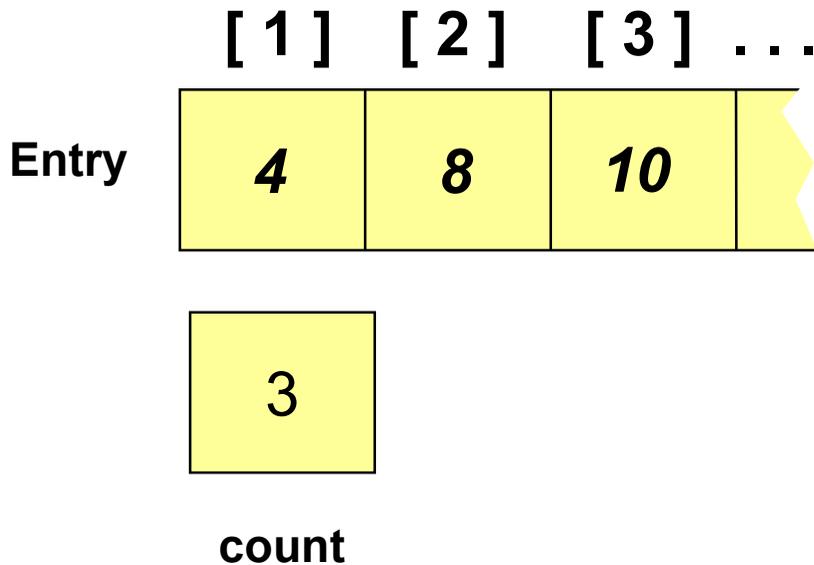


Quais valores serão armazenados em **Entry** e **count** depois que a chamada de procedimento termina?

# Operações Básicas: Delete

```
void List::Delete(int p, int &x)
```

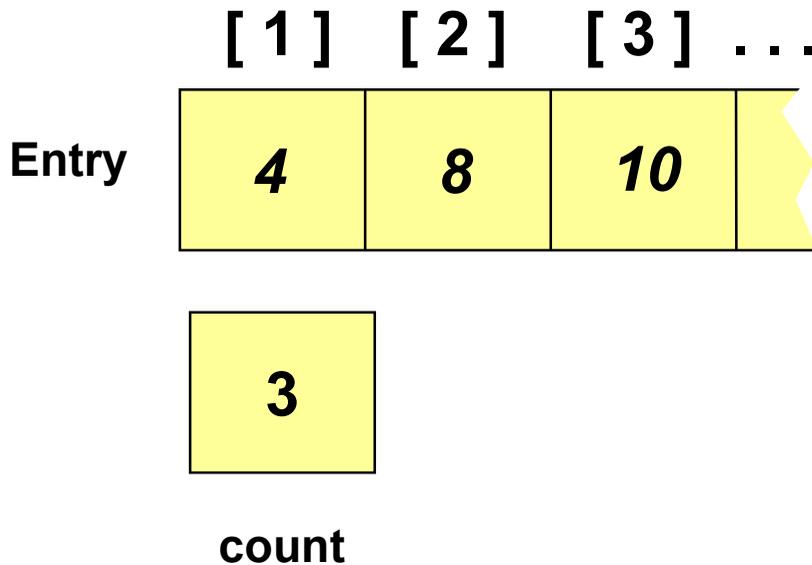
Depois da chamada a Delete(1,x),  
nós teremos esta lista:



# Operações Básicas: Delete

```
void List::Delete(int p, int &x)
```

Antes de remover, é conveniente verificar se a lista não está vazia...

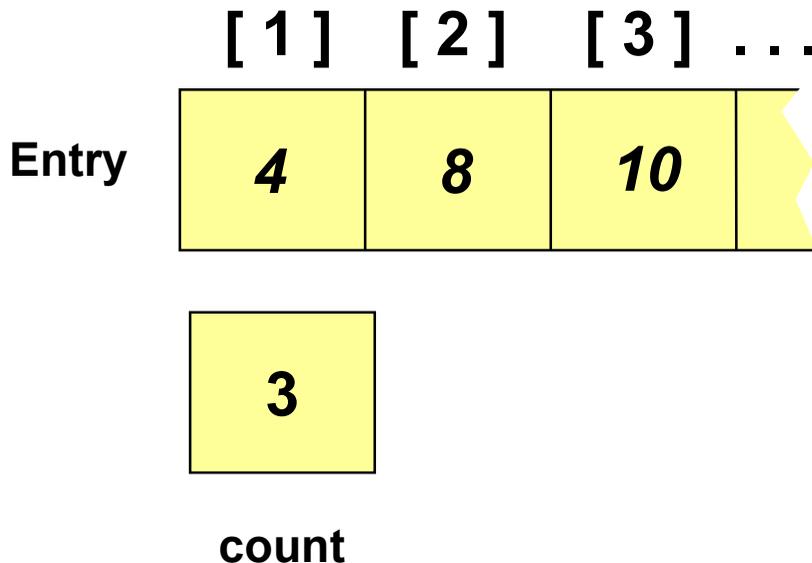


```
void List::Delete(int p,int &x)
{ int i;
  if (Empty())
  { cout << "Lista Vazia";
    abort();
  }
  ...
}
```

# Operações Básicas: Delete

```
void List::Delete(int p, int &x)
```

...e se a posição de remoção é válida

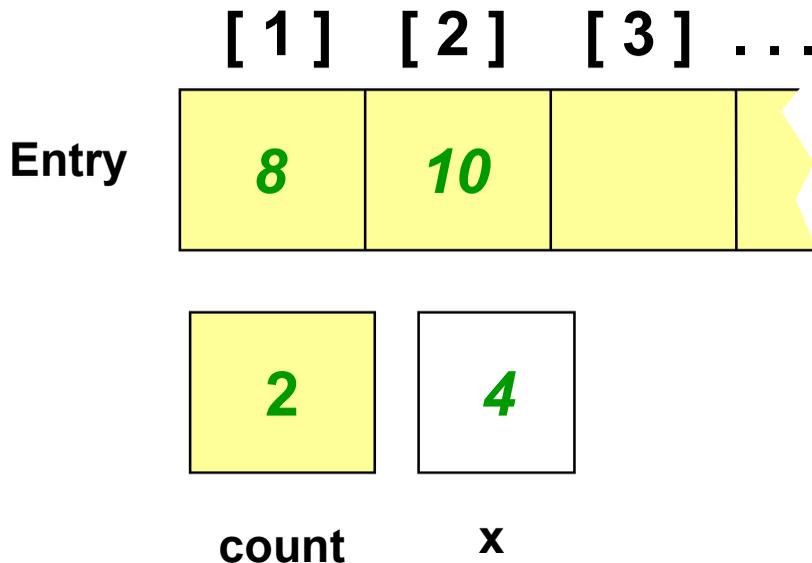


```
void List::Delete(int p,int &x)
{ int i;
  if (Empty())
  { cout << "Lista Vazia";
    abort();
  }
  if (p < 1 || p > count)
  { cout << "Posição inválida";
    abort();
  }
  ...
}
```

# Operações Básicas: Delete

```
void List::Delete(int p, int &x)
```

Agora basta recuperar o elemento **x** e deslocar os elementos subsequentes



```
void List::Delete(int p,int &x)
{ int i;
  if (Empty())
  { cout << "Lista Vazia";
    abort();
  }
  if (p < 1 || p > count)
  { cout << "Posição inválida";
    abort();
  }
  x = Entry[p];
  for(i=p; i<count; i++)
    Entry[i] = Entry[i+1];
  count = count - 1;
}
```

# Exercícios

---

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

# Solução Clear/Size

---

```
void List::Clear()
{
    count = 0;
}
```

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

# Solução Retrieve/Replace

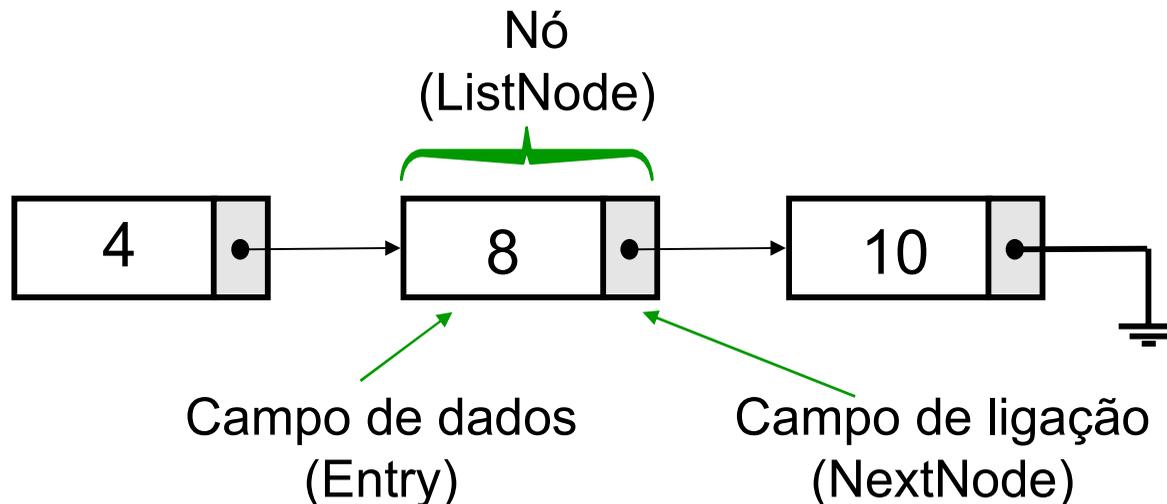
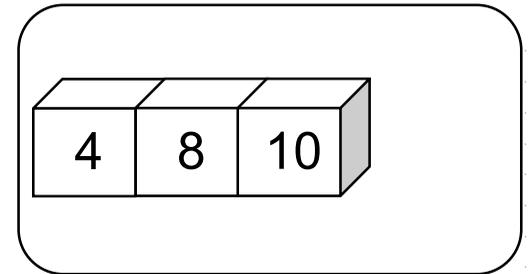
---

```
void List::Retrieve(int p, int
    &x)
{ if(p < 1 || p > count)
  { cout << "Posição inválida";
    abort();
  }
  x = Entry[p];
}
```

```
void List::Replace(int p, int x)
{ if(p < 1 || p > count)
  { cout << "Posição inválida";
    abort();
  }
  Entry[p] = x;
}
```

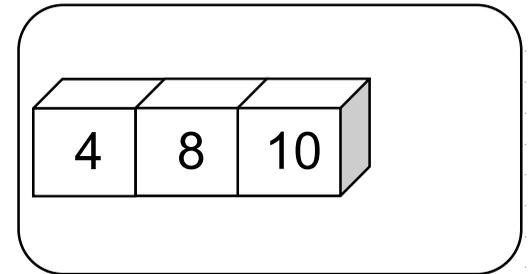
# Implementação Encadeada

- ❑ 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**)

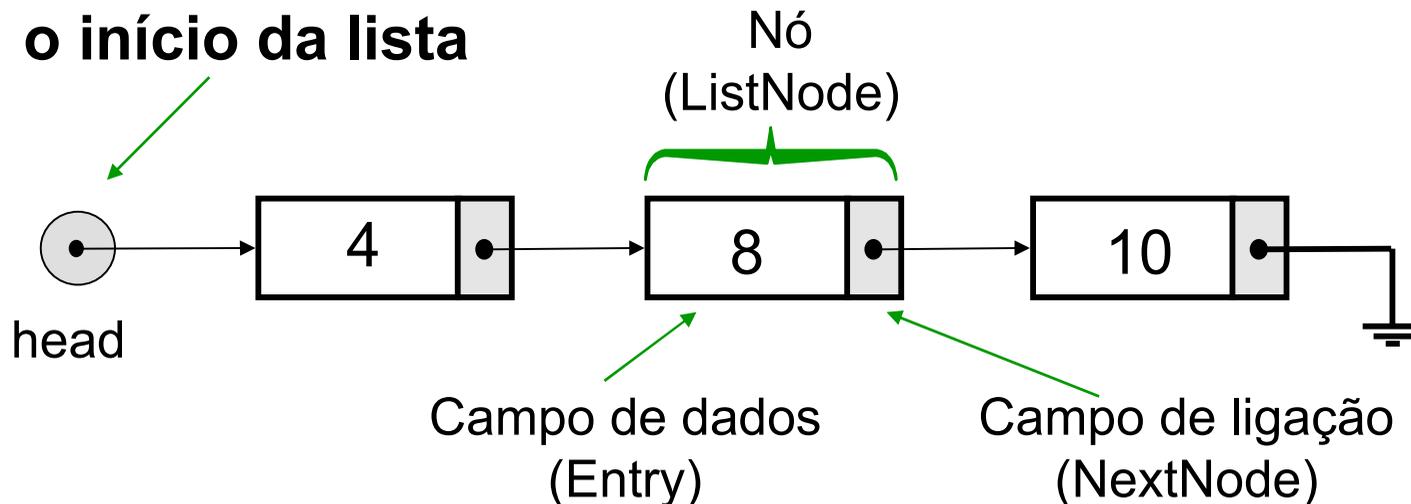


# Implementação Encadeada

- ❑ Nós precisamos armazenar o início da lista...

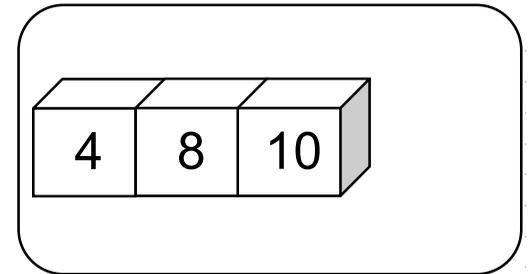


**Um ponteiro armazena o início da lista**

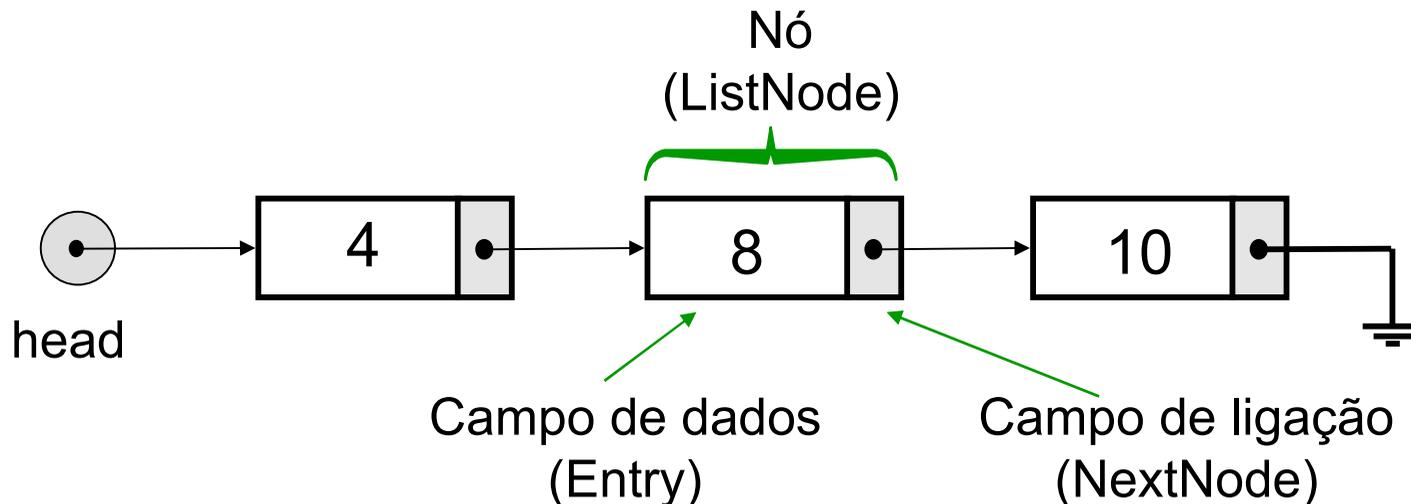


# Implementação Encadeada

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



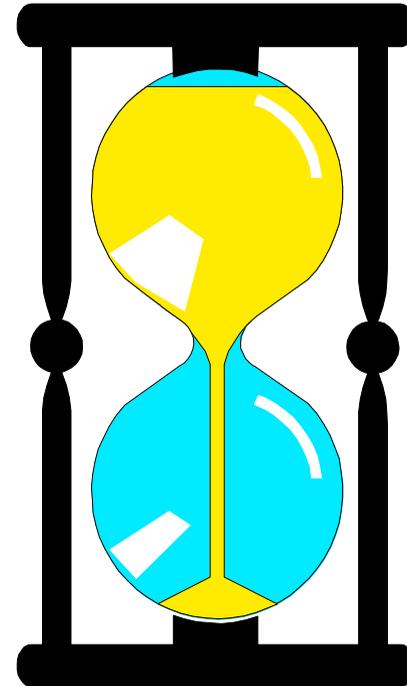
**3** count



# 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

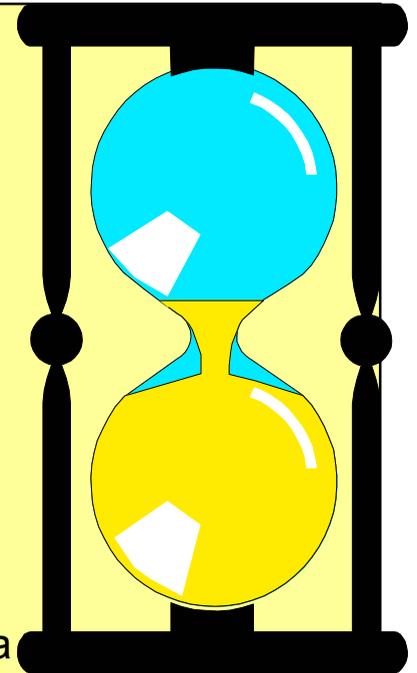


Você tem 5 minutos para escrever a declaração

# Uma Solução

```
class List
{ public:
  List();
  ~List();
  void Insert(int p, int x);
  void Delete(int p, 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; // início da lista
  int count;       // número de elementos
};
```



# Uma Solução

```
class List
{ public:
    List();
    ~List();
    void Insert(int p, int x);
    void Delete(int p, 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;

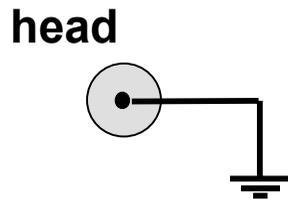
    // declaração de campos
    ListPointer head; // início da lista
    int count;       // número de elementos
};
```

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

# Construtor

```
List::List()
```

A Lista deve iniciar vazia...

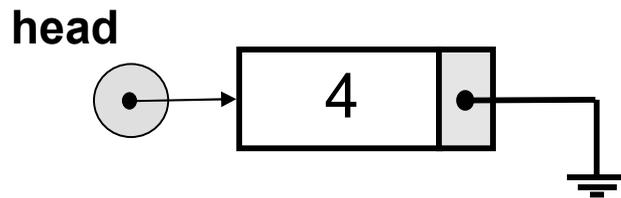


```
List::List()
{
    head = NULL;
    count = 0;
}
```

# Destruidor

```
List::~~List()
```

Podemos fazer uma chamada ao método **Clear**, que libera o espaço alocado pela lista

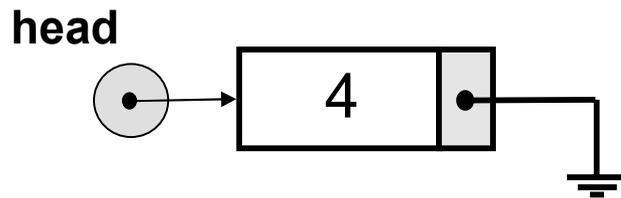


```
List::~~List()  
{  
    Clear();  
}
```

# Destruidor

```
List::~~List()
```

Alternativamente, 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!



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

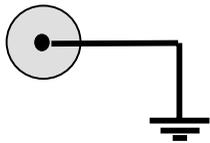
  while (head != NULL)
  { q = head;
    head = head->NextNode;
    delete q;
  }
}
```

# Status: Empty

```
bool List::Empty()
```

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

head

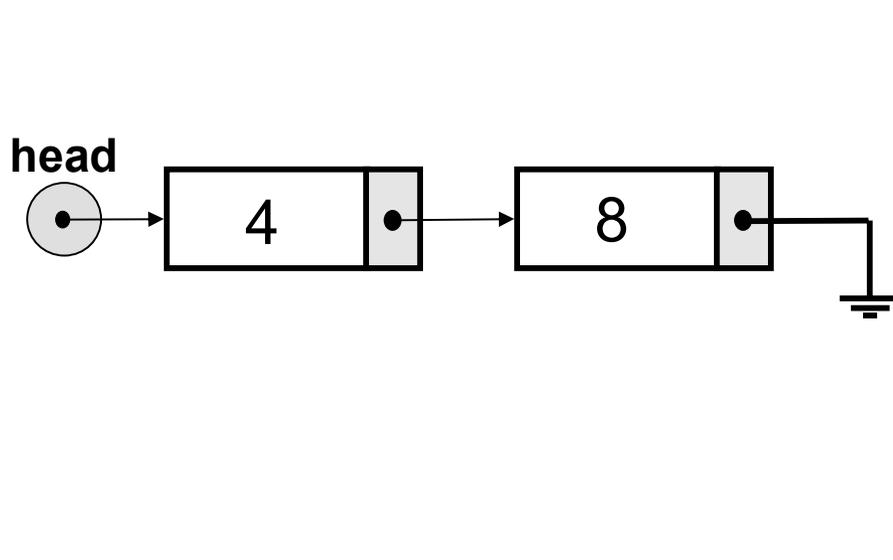


```
bool List::Empty()
{
    return (head == NULL);
}
```

# Status: Full

```
bool List::Full()
```

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



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

# Encontrando uma Posição na Lista

---

- ❑ Desde que devemos ser capazes de substituir a implementação encadeada diretamente por uma implementação contígua, torna-se necessário um método que aceita como entrada a posição — um inteiro indicando um índice na lista — e retorna um ponteiro (`ListPointer`) para o nó correspondente na lista
- ❑ O método privado **SetPosition** inicia no primeiro elemento da lista e a atravessa até encontrar o nó desejado

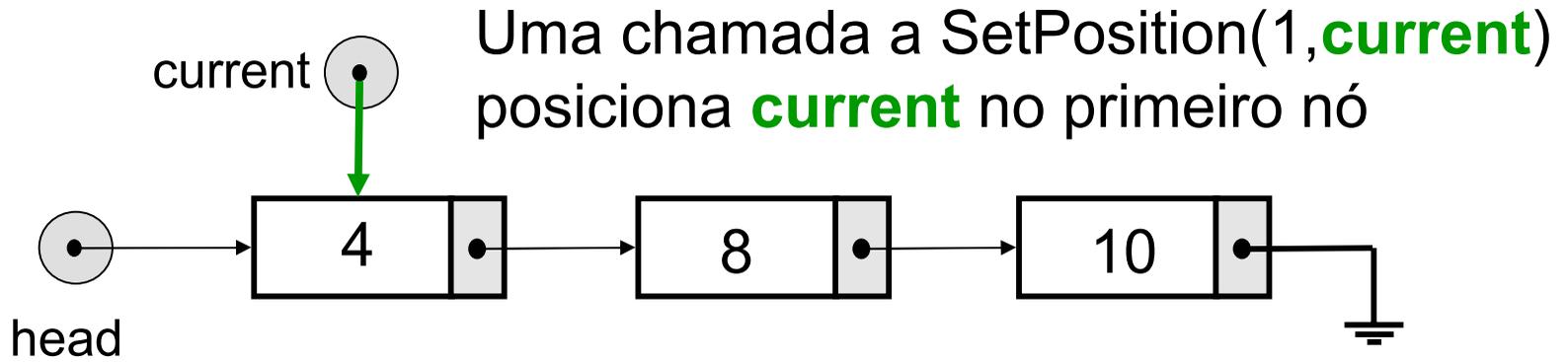
# Encontrando uma Posição na Lista

```
class List
{ public:
    List();
    ~List();
    void Insert(int p, int x);
    void Delete(int p, 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;

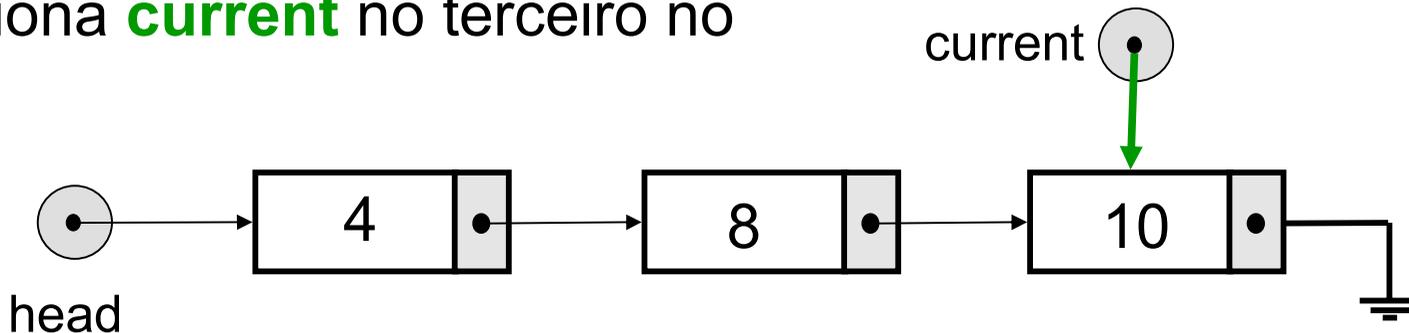
    // declaração de campos
    ListPointer head; // início da lista
    int count;       // número de elementos

    // métodos privados
    void SetPosition(int p, ListPointer &current);
};
```

# SetPosition: Exemplo



Uma chamada a `SetPosition(3, current)` posiciona **current** no terceiro nó



# Encontrando uma Posição na Lista

---

```
void List::SetPosition(int p, ListPointer &current)
// pré-condição: p é uma posição válida na lista
// pós-condição: o ponteiro current aponta para o nó na lista
//                com posição p
{ int i;

  if (p < 1 || p > count+1)
  { cout << "Posição inválida";
    abort();
  }
  current = head;
  for(i=2; i<=p; i++)
    current = current->NextNode;
}
```

# Encontrando uma Posição na Lista

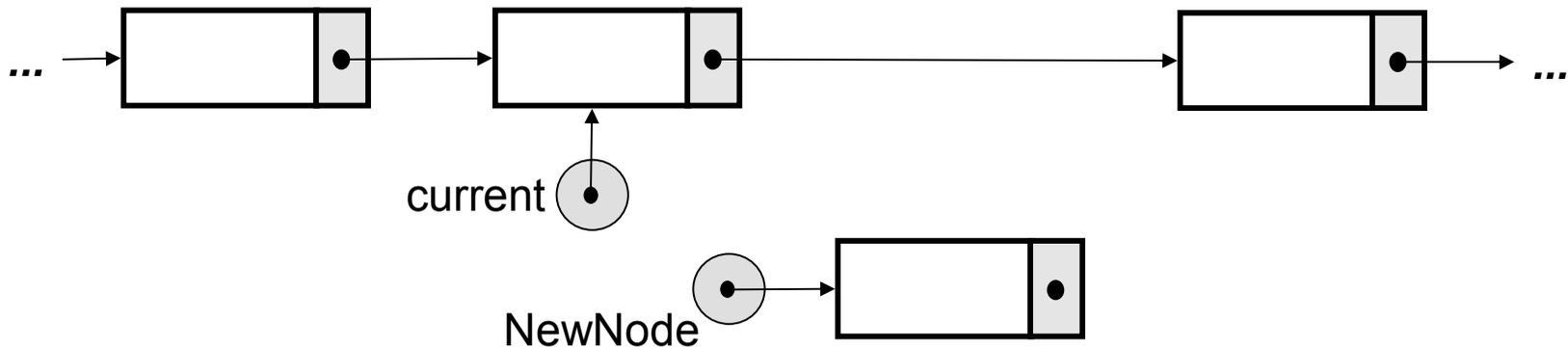
```
void List::SetPosition(int p, ListPointer &current)
// pré-condição: p é uma posição válida na lista
// pós-condição: o ponteiro current aponta para o nó na lista
//                com posição p
{ int i;

  if (p < 1 || p > count+1)
  { cout << "Posição inválida";
    abort();
  }
  current = head;
  for(i=2; i<=p; i++)
    current = current->NextNode;
}
```

Este fragmento de código pode ser removido, considerando que todos os métodos que o chamam já efetuam este teste

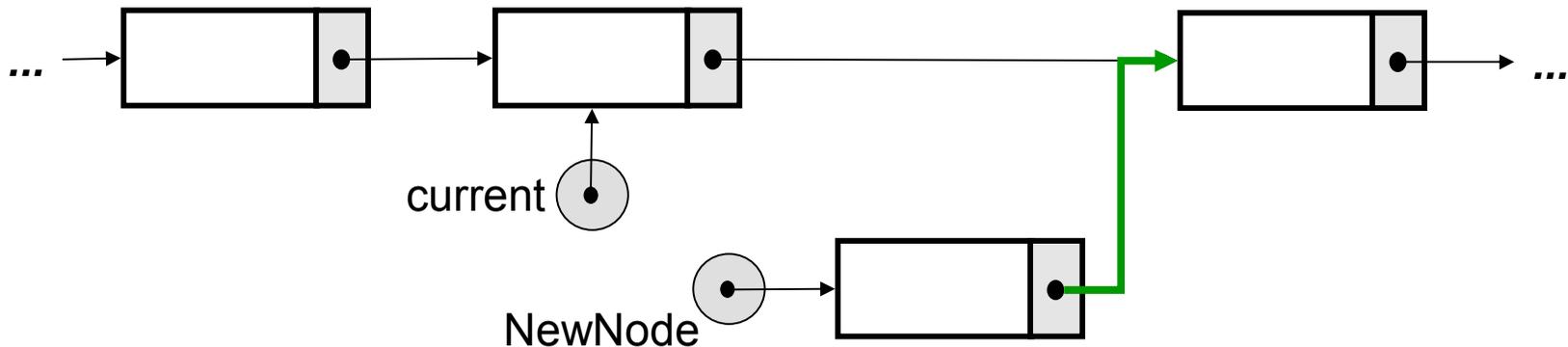
# Operações Básicas: Insert

- ❑ Se temos um novo nó a ser inserido na lista, então é necessário encontrar um ponteiro anterior à posição que o novo nó deve ser inserido
- ❑ Seja **NewNode** um ponteiro para o novo nó a ser inserido e **current** um ponteiro para o nó precedente



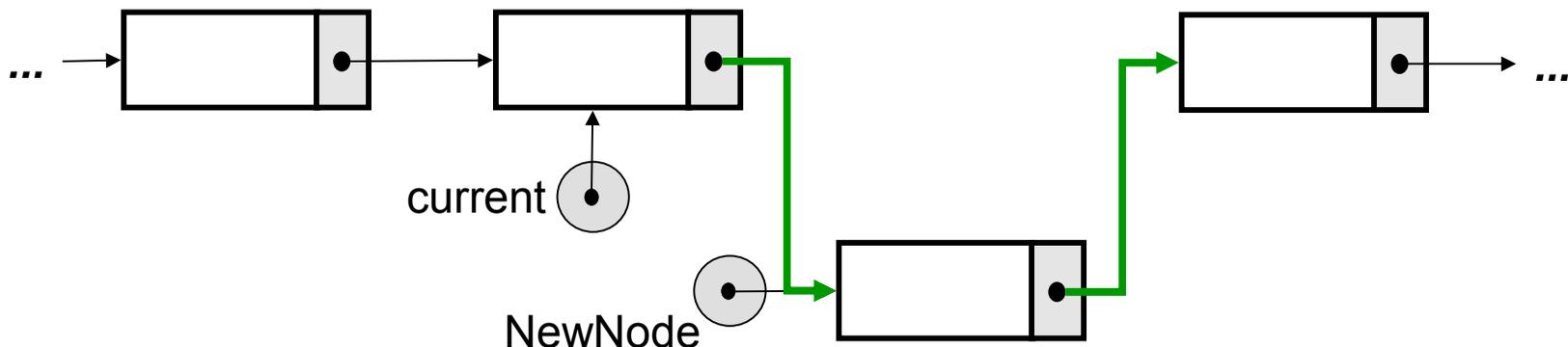
# Operações Básicas: Insert

- ❑ Se temos um novo nó a ser inserido na lista, então é necessário encontrar um ponteiro anterior à posição que o novo nó deve ser inserido
- ❑ Seja **NewNode** um ponteiro para o novo nó a ser inserido e **current** um ponteiro para o nó precedente, então o fragmento de código seguinte efetua a inserção:
  - `NewNode->NextNode = current->NextNode;`
  - ...



# Operações Básicas: Insert

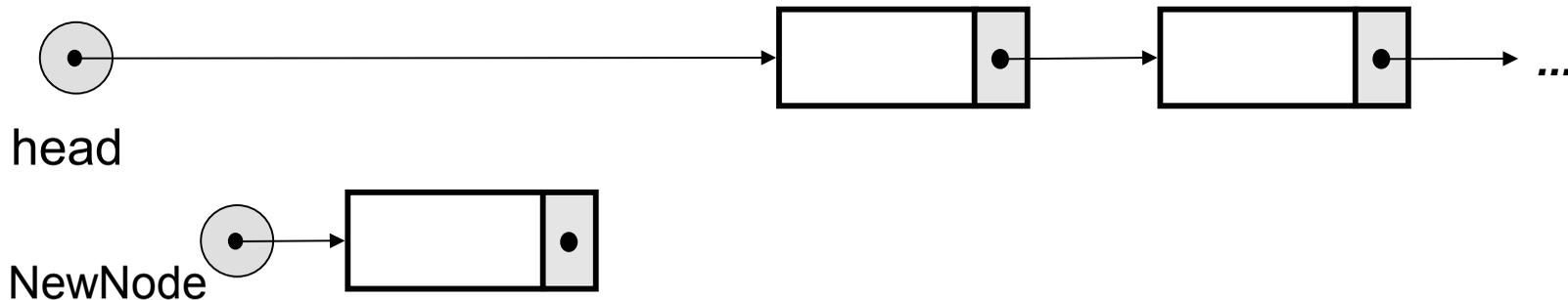
- ❑ Se temos um novo nó a ser inserido na lista, então é necessário encontrar um ponteiro anterior à posição que o novo nó deve ser inserido
- ❑ Seja **NewNode** um ponteiro para o novo nó a ser inserido e **current** um ponteiro para o nó precedente, então o fragmento de código seguinte efetua a inserção:
  - `NewNode->NextNode = current->NextNode;`
  - `current->NextNode = NewNode;`



# Operações Básicas: Insert

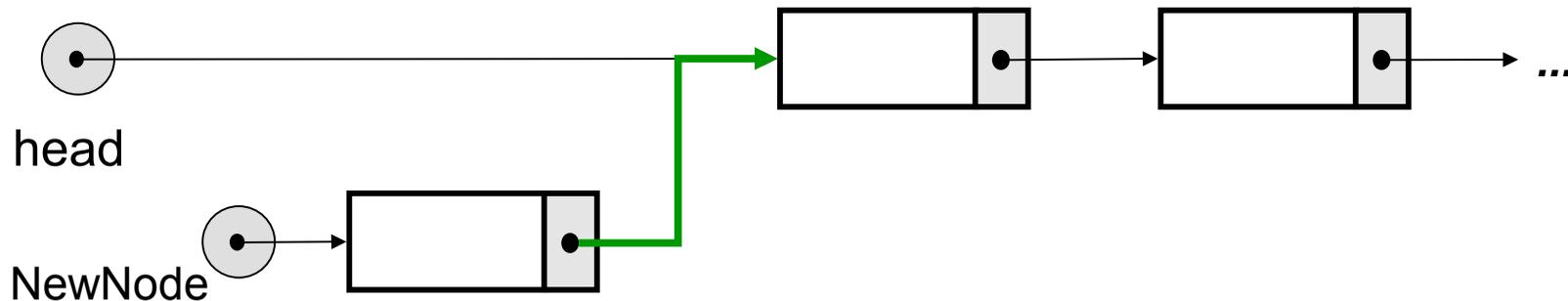
---

- Entretanto, se a inserção ocorrer no início da lista...



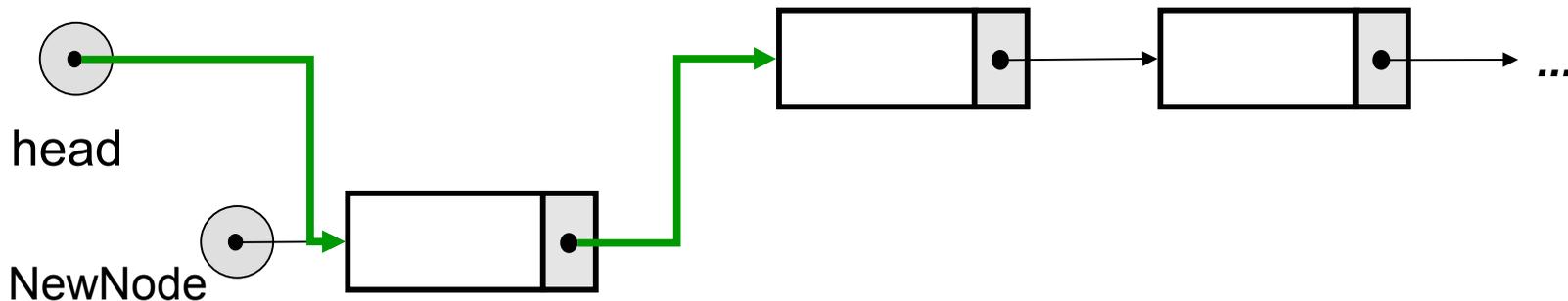
# Operações Básicas: Insert

- Entretanto, se a inserção ocorrer no início da lista, o código é:
  - `SingleNode->NextNode = head;`
  - ...



# Operações Básicas: Insert

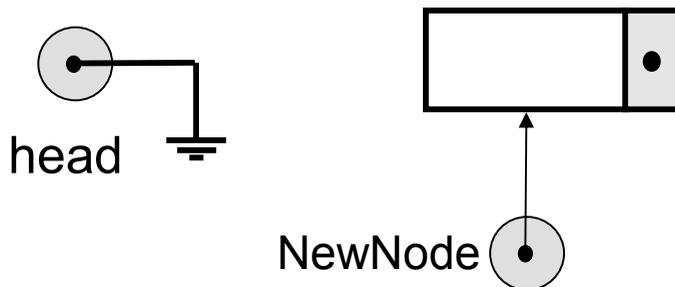
- Entretanto, se a inserção ocorrer no início da lista, o código é:
  - `SingleNode->NextNode = head;`
  - `head = NewNode;`



# Operações Básicas: Insert

---

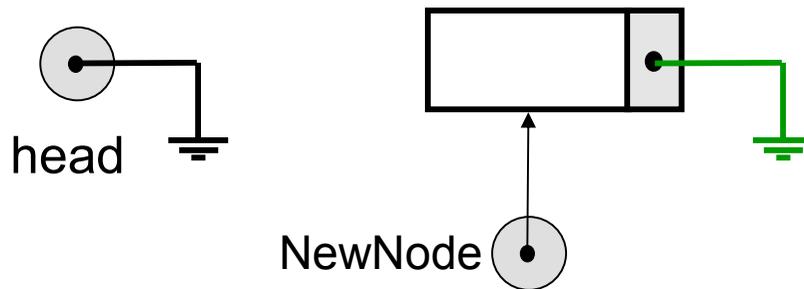
- ❑ Questão: isso funciona se a lista estiver vazia?
  - `SingleNode->NextNode = head;`
  - `head = NewNode;`



# Operações Básicas: Insert

---

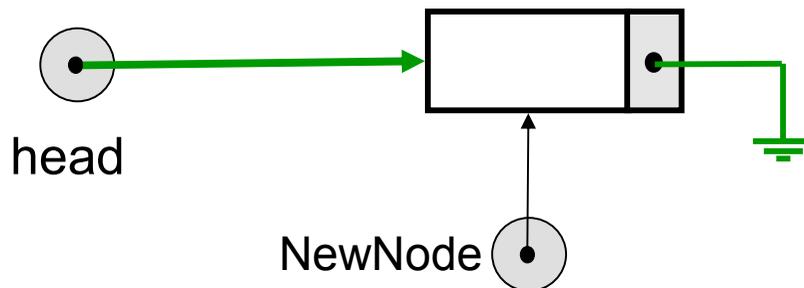
- ❑ Questão: isso funciona se a lista estiver vazia?
  - `SingleNode->NextNode = head;`
  - `head = NewNode;`



# Operações Básicas: Insert

---

- ❑ Questão: isso funciona se a lista estiver vazia?
  - `SingleNode->NextNode = head;`
  - `head = NewNode;`



# Operações Básicas: Insert

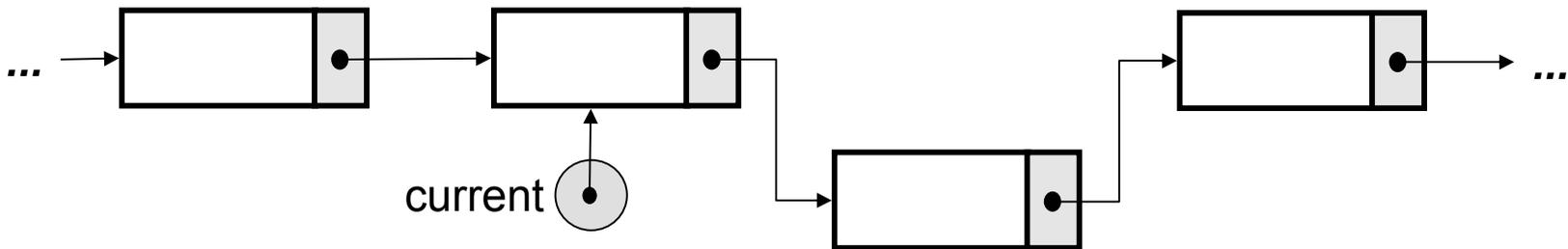
---

```
void List::Insert(int p, int x)
{ ListPointer NewNode, current;

  if (p < 1 || p > count+1)
  { cout << "Posição inválida";
    abort();
  }
  NewNode = new ListNode;
  NewNode->Entry = x;
  if(p == 1)
  { NewNode->NextNode = head;
    head = NewNode;
  }
  else
  { SetPosition(p-1,current);
    NewNode->NextNode = current->NextNode;
    current->NextNode = NewNode;
  }
  count++;
}
```

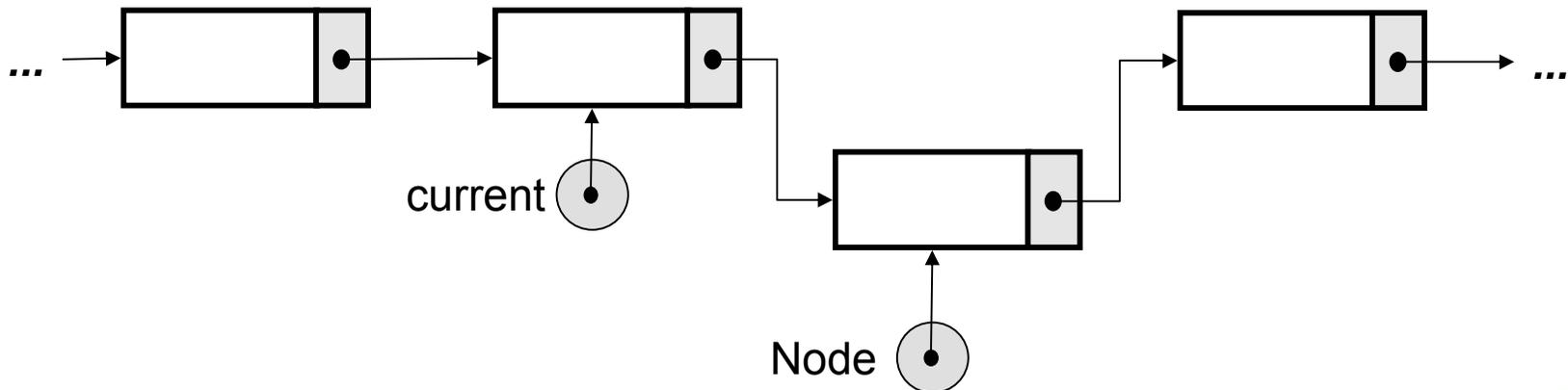
# Operações Básicas: Delete

- ❑ Se temos um nó a ser removido da lista, então é necessário encontrar um ponteiro anterior à ele
- ❑ Seja **Node** um ponteiro para o nó a ser removido e **current** um ponteiro para o nó precedente



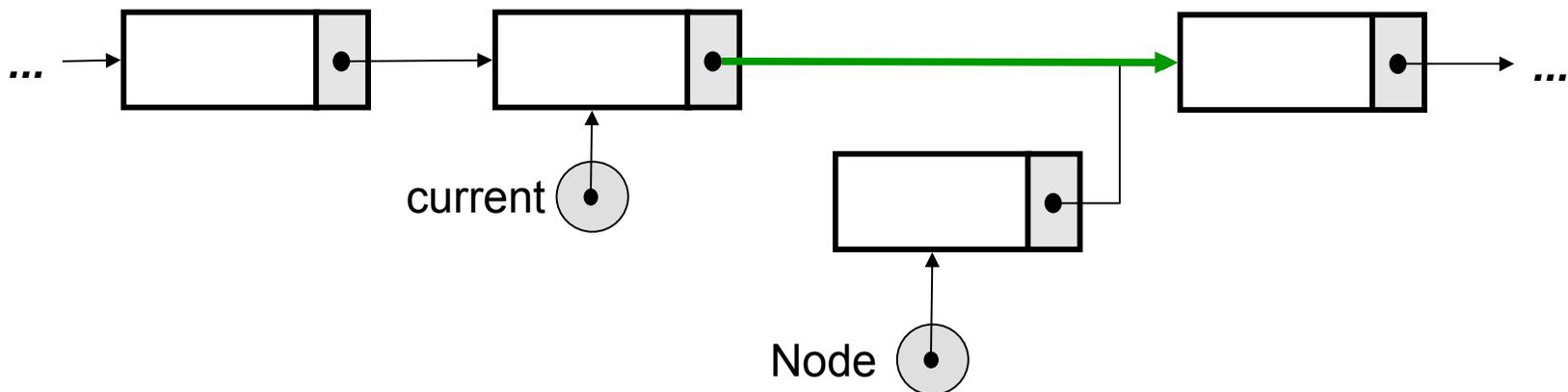
# Operações Básicas: Delete

- ❑ Se temos um nó a ser removido da lista, então é necessário encontrar um ponteiro anterior à ele
- ❑ Seja **Node** um ponteiro para o nó a ser removido e **current** um ponteiro para o nó precedente, ou seja:
  - `Node = current->NextNode;`



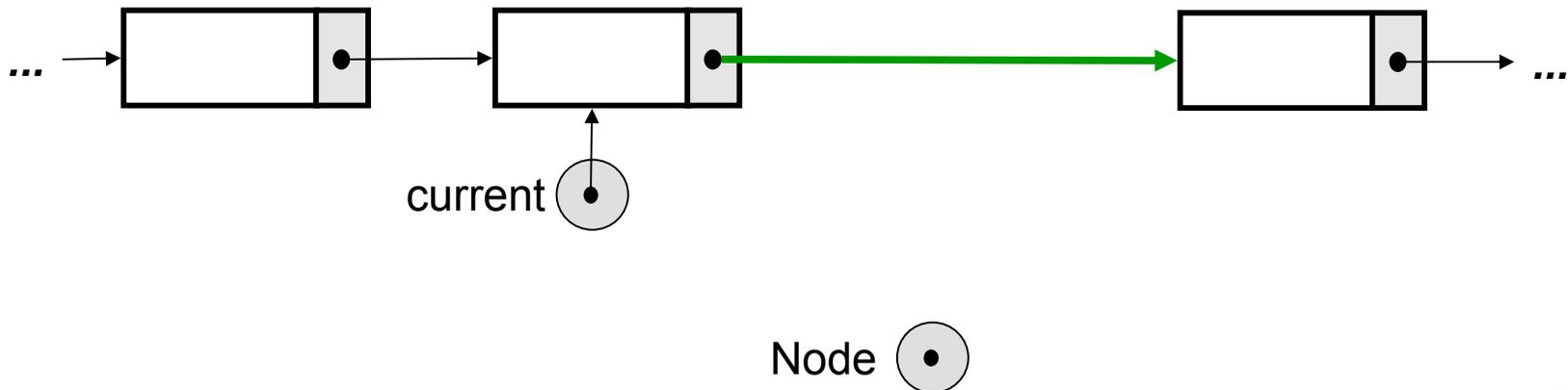
# Operações Básicas: Delete

- ❑ Se temos um nó a ser removido da lista, então é necessário encontrar um ponteiro anterior à ele
- ❑ Seja **Node** um ponteiro para o nó a ser removido e **current** um ponteiro para o nó precedente, ou seja:
  - `Node = current->NextNode;`
- ❑ A remoção ocorre efetuando as operações:
  - `current->NextNode = Node->NextNode;`
  - ...



# Operações Básicas: Delete

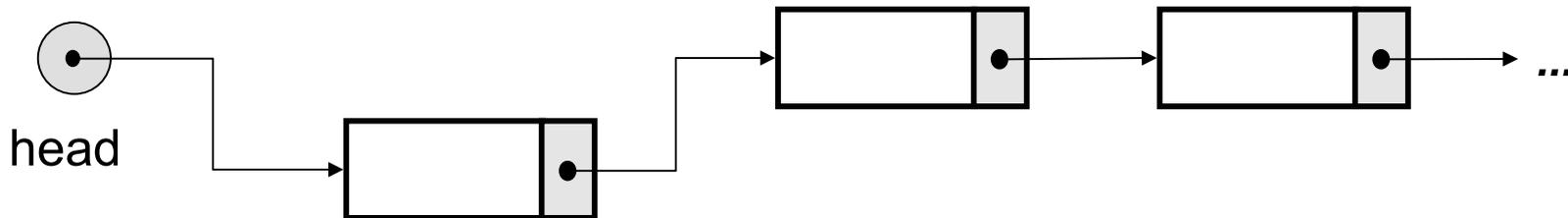
- ❑ Se temos um nó a ser removido da lista, então é necessário encontrar um ponteiro anterior à ele
- ❑ Seja **Node** um ponteiro para o nó a ser removido e **current** um ponteiro para o nó precedente, ou seja:
  - `Node = current->NextNode;`
- ❑ A remoção ocorre efetuando as operações:
  - `current->NextNode = Node->NextNode;`
  - `delete Node;`



# Operações Básicas: Delete

---

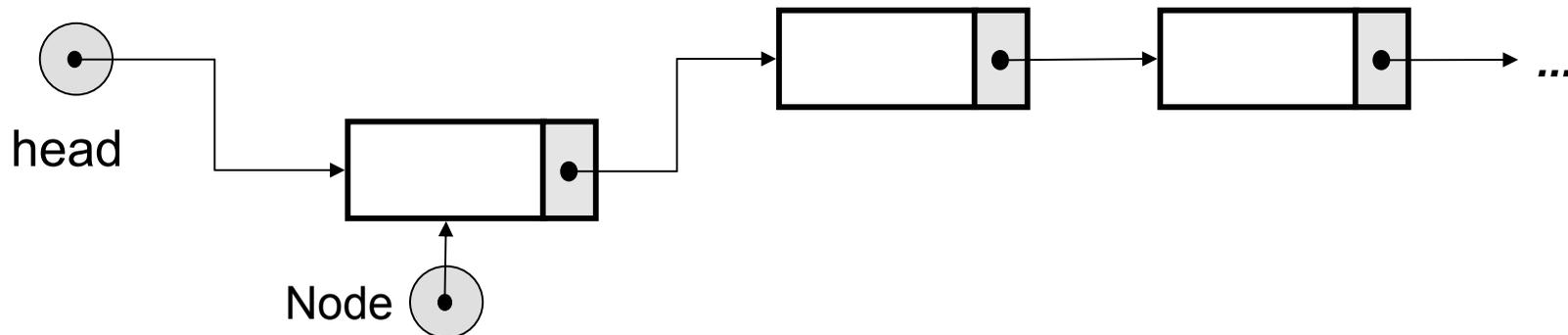
- ❑ Entretanto, se a remoção ocorrer no início da lista...



# Operações Básicas: Delete

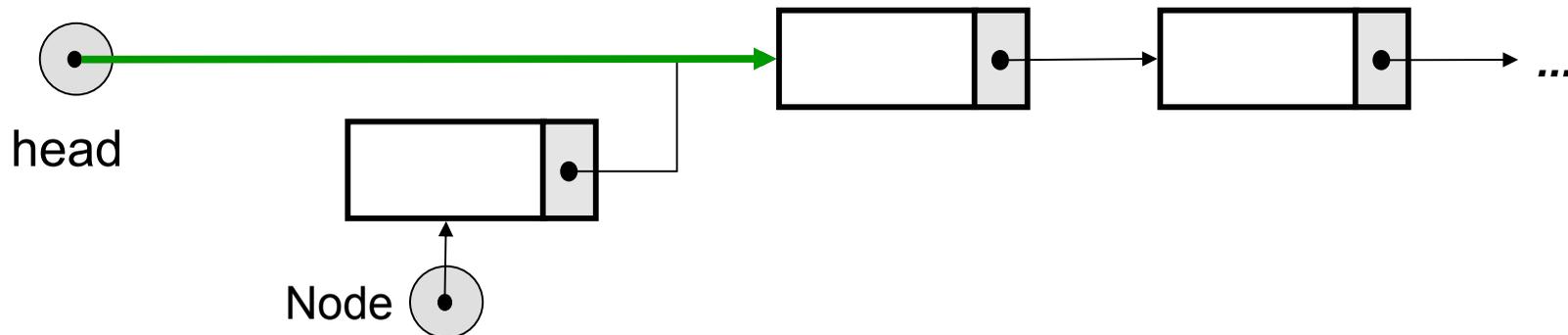
□ Entretanto, se a remoção ocorrer no início da lista

- Node = head;
- ...



# Operações Básicas: Delete

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

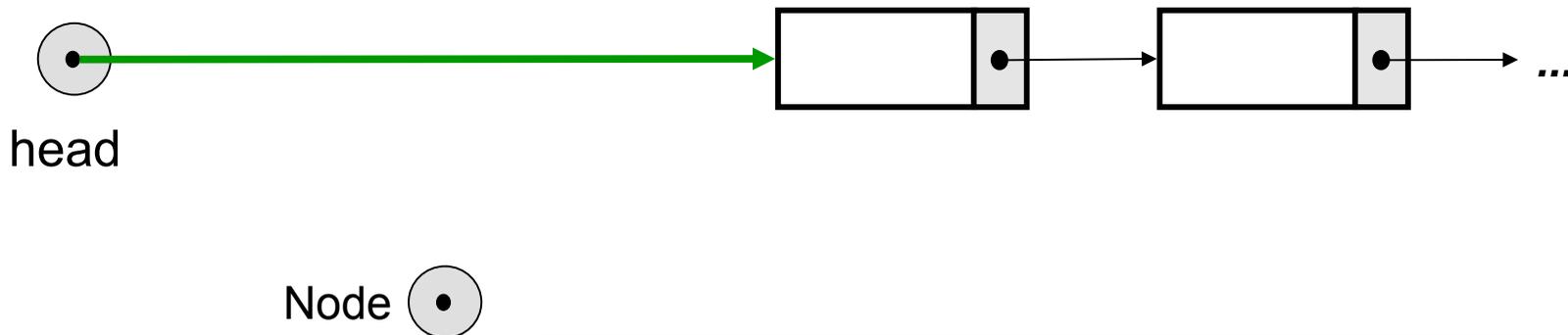


# Operações Básicas: Delete

---

□ Entretanto, se a remoção ocorrer no início da lista

- Node = head;
- head = Node->NextNode;
- delete Node;



# Operações Básicas: Delete

---

```
void List::Delete(int p, int &x)
{ ListPointer Node, current;

  if (p < 1 || p > count)
  { cout << "Posição inválida";
    abort();
  }
  if(p == 1)
  { Node = head;
    head = Node->NextNode;
  }
  else
  { SetPosition(p-1,current);
    Node = current->NextNode;
    current->NextNode = Node->NextNode;
  }
  x = Node->Entry;
  delete Node;
  count = count - 1;
}
```

# Exercícios

---

- Implemente seguintes operações em listas
  - Clear()
  - Size()
  - Retrieve()
  - Replace()

# Solução Clear/Size

---

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

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

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

# Solução Retrieve/Replace

---

```
void List::Retrieve(int p, int
    &x)
{ ListPointer q;

    if(p < 1 || p > count)
    { cout << "Posição inválida";
      abort();
    }
    SetPosition(p,q);
    x = q->Entry;
}
```

```
void List::Replace(int p, int x)
{ ListPointer q;

    if(p < 1 || p > count)
    { cout << "Posição inválida";
      abort();
    }
    SetPosition(p,q);
    q->Entry = x;
}
```

# Exercícios Adicionais

---

- Assumindo uma lista de inteiros, implemente as seguintes operações
  - Minimum()/Maximum()
    - ❖ Retorna posição **p** que o elemento de menor (maior) valor **x** se encontra na lista; caso não encontre, retorna **p** igual a zero
  - Reverse()
    - ❖ Versão que inverte a posição dos elementos da lista
  - Reverse()
    - ❖ Versão que inverte a posição dos elementos criando uma nova lista; a lista original permanece intacta
  - Copy()
    - ❖ Copia elementos da lista atual para uma nova lista; a lista original permanece intacta (sem uso de nó adicional temporário)
  - Copy()
    - ❖ Copia elementos da lista atual para uma nova lista; a lista original permanece intacta (utilizando nó adicional temporário)
  - Sort()
    - ❖ Ordena elementos da lista em ordem crescente (sem uso de nó adicional temporário)
  - Sort()
    - ❖ Ordena elementos da lista em ordem crescente (utilizando nó adicional temporário)

# Minimum

---

```
void List::Minimum(int &p, int &x)
// pré: Lista criada e não vazia
// pós: Retorna posição p que o elemento de menor valor x se encontra
//      na lista; caso não encontre, retorna p igual a zero
{ ListPointer q=head;
  int pos;

  if(Empty())
    p = 0; // lista vazia, não existe mínimo
  else
  { x = q->Entry;      p = 1;      // assumir 1o. elemento como o menor
    q = q->NextNode;  pos = 2;    // procurar do 2o. elemento até o final da lista
    while (q != NULL)
    { if(q->Entry < x)
      { x = q->Entry;          // achou novo mínimo...
        p = pos;              // ...anotar a posição dele
      }
      q = q->NextNode;  pos++;    // próximo elemento da lista
    }
  }
}
```

# Reverse (Mesma Lista)

---

```
void List::Reverse()
// pré: Lista criada
// pós: Inverte a posição dos elementos da lista
{ ListPointer p=head, q=NULL;

  while(p != NULL)
  { head = p;
    p = p->NextNode;
    head->NextNode = q;
    q = head;
  }
}
```

# Reverse (Nova Lista)

---

```
void List::Reverse(List &L)
// pré: Listas criadas (this e L)
// pós: Inverte a posição dos elementos da lista criando lista L; a lista
//      original permanece intacta
{ ListPointer p=head, r;

  L.Clear();
  while(p != NULL)
  { r = new ListNode;
    r->Entry = p->Entry;
    r->NextNode = L.head;
    L.head = r;
    p = p->NextNode;
  }
  L.count = count;
}
```

# Copy

---

```
void List::Copy(List &L)
// pré: Listas criadas (this e L)
// pós: Copia elementos da lista atual (this) para lista L; a lista original permanece intacta
{ ListPointer p=head, q=NULL, NewNode;

  L.Clear();
  while(p != NULL)
  { NewNode = new ListNode;
    NewNode->Entry = p->Entry;
    if(q == NULL)
      L.head = NewNode;
    else
      q->NextNode = NewNode;
    q = NewNode;
    p = p->NextNode;
  }
  if (q != NULL)
    q->NextNode = NULL;
  L.count = count;
}
```

# Copy (Com Nó Adicional Temporário)

---

```
void List::Copy(List &L)
// pré: Listas criadas (this e L)
// pós: Copia elementos da lista atual (this) para lista L; a lista original permanece intacta
{ ListPointer p=head, q, NewNode;

  L.Clear();
  q = new ListNode;           // nó adicional
  L.head = q;
  while(p != NULL)
  { NewNode = new ListNode;
    NewNode->Entry = p->Entry;
    q->NextNode = NewNode;
    q = NewNode;
    p = p->NextNode;
  }
  q->NextNode = NULL;
  q = L.head;                 // remover nó adicional
  L.head = L.head->NextNode;
  delete q;
  L.count = count;
}
```

---

# Insertion Sort

---

```
void List::Sort()
// pré: Lista criada
// pós: Ordena elementos da lista usando inserção direta
{ ListPointer p,           // lista não ordenada
  NewHead, q, r;         // lista ordenada

  NewHead = NULL;
  while(head != NULL)
  { p = head;           head = head->NextNode;
    q = NULL;          r = NewHead;           // q fica um nó atrás de r
    while(r != NULL && r->Entry < p->Entry)
    { q = r;           r = r->NextNode;
    }
    if(r == NewHead)
      NewHead = p;
    else
      q->NextNode = p;
      p->NextNode = r;
  }
  head = NewHead;
}
```

# Insertion Sort

## (Com Nó Adicional Temporário)

---

```
void List::Sort()
// pré: Lista criada
// pós: Ordena elementos da lista usando inserção direta
{ ListPointer p,           // lista não ordenada
    NewHead, q, r;        // lista ordenada

    NewHead = new ListNode;  NewHead->NextNode = NULL;
    while(head != NULL)
    { p = head;          head = head->NextNode;
      q = NewHead;      r = NewHead->NextNode;    // q fica um nó atrás de r
      while(r != NULL && r->Entry < p->Entry)
      { q = r;          r = r->NextNode;
      }
      q->NextNode = p;
      p->NextNode = r;
    }
    head = NewHead->NextNode;
    delete NewHead;
}
```

# Busca

---

- ❑ A busca de um elemento **x** em uma lista é uma operação que ocorre com bastante frequência
- ❑ Diferentemente do que ocorre com a utilização de vetores, a busca neste caso deve ser puramente seqüencial
- ❑ A busca termina ao encontrar o elemento desejado ou quando o final da lista for atingido, o que implica em *duas condições lógicas*

# Busca

---

## ❑ Implementação Contígua

- Assuma o início da lista **q=1**:
  - ❖ while (q <= count && Entry[q] != x)  
q++;
- **q > count** implica que **Entry[q]** não possui valor significativo (“lixo”) ou que ele não existe e, portanto, que a expressão **Entry[q] != x** está indefinida
- Logo, a ordem em que estes dois testes são efetuados é de suma importância

## ❑ Implementação Encadeada Dinâmica

- Assuma que o início da lista seja apontado pelo ponteiro **q**, ou seja, inicialmente **q=head**:
  - ❖ while (q != NULL && q->Entry != x)  
q = q->NextNode;
- **q == NULL** implica que **q->** não existe e, portanto, que a expressão **q->Entry != x** está indefinida
- Novamente, a ordem em que estes dois testes são efetuados é de suma importância

# Busca (Implementação Contígua)

---

```
int List::Search(int x)
// pré: Lista criada
// pós: Retorna posição que o elemento x encontra-se na
//      lista; caso não encontre, retorna zero
{ int p=1;

  while (p <= count && Entry[p] != x)
    p++;
  return (p > count ? 0 : p);
}
```

# Busca (Implementação Encadeada)

---

```
int List::Search(int x)
// pré: Lista criada
// pós: Retorna posição que o elemento x encontra-se na
//      lista; caso não encontre, retorna zero
{ int p=1;
  ListPointer q=head;

  while (q != NULL && q->Entry != x)
  { q = q->NextNode;
    p++;
  }
  return (q == NULL ? 0 : p);
}
```

# Busca

---

- ❑ O algoritmo Busca (Encadeada) apresenta fortes semelhanças com o algoritmo Busca (Contígua), ou seja, com o algoritmo de busca em vetores
- ❑ De fato, um vetor é uma lista linear para o qual a técnica de ligação ao sucessor é deixada implícita
- ❑ As listas lineares com apontadores explícitos oferecem maior flexibilidade devendo, portanto, ser utilizadas quando essa flexibilidade adicional se faz necessária
- ❑ Assim como na busca seqüencial em vetores, a utilização de um elemento **sentinela** simplifica e agiliza os algoritmos, como veremos nas próximas aulas

# Resumo

---

- ❑ O ADT Lista é uma seqüência de elementos, que podem ser inseridos, removidos ou alterados em qualquer posição
- ❑ Pilhas e Filas são Listas restritas, ou seja, usando a especificação de Lista, podemos implementar Pilhas e Filas