Universidade Federal da Paraíba

Centro de Informática

Departamento de Informática

Estrutura de Dados
Listas

- Tiago Maritan
- tiago@ci.ufpb.br

Conteúdos Abordados

- O Conceito de Listas
- Listas com Representação Sequencial
- Listas com Representação Dinâmica
  - Listas Simplesmente Encadeadas
  - Listas Duplamente Encadeadas
  - Listas Circulares

Listas

- O que são Listas?
  - **Estruturas de dados lineares** que agrupam informações referentes a um **conjunto de elementos relacionados**.

- Exemplos:
  - Lista de clientes de uma agência bancária;
  - Lista de setores de disco a serem acessados por um SO;
  - Lista de pacotes a serem transmitidos em um nó de uma rede de comutação de pacotes.

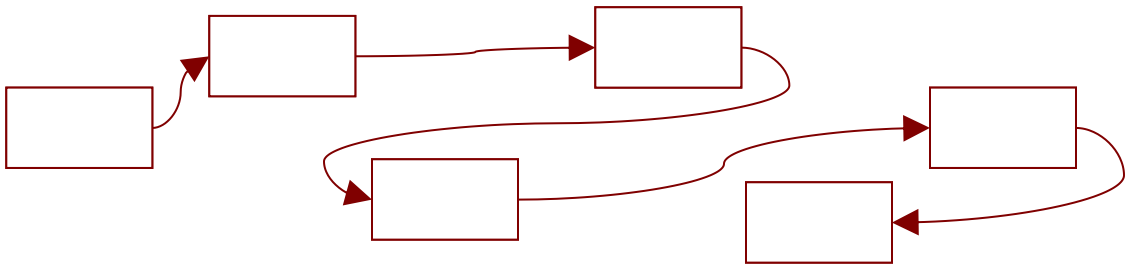
Listas

- Conjunto de operações (interface):
  - Criar uma lista vazia;
  - Verificar se uma lista está vazia;
  - Verificar se uma lista está cheia;
  - **Inserir um novo elemento após (ou antes) de uma determinada posição na lista;**
  - **Remover um elemento de uma determinada posição na lista;**
  - **Exibir os elementos de uma lista, etc.**

Formas de Representação de Listas

- **Alocação Sequencial**
  - Elementos dispostos em posições contíguas de memória

- **Alocação Encadeada**
  - Elementos dispostos aleatoriamente na memória, encadeados por ponteiros



Listas com Representação Sequencial

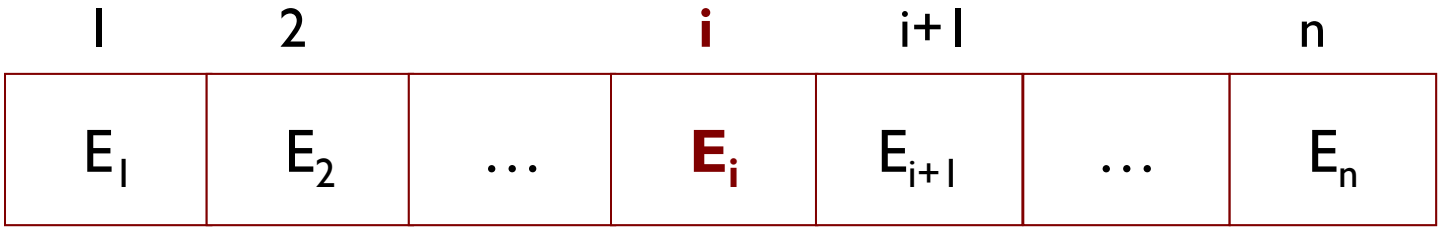
- Conjunto de registros (elementos), onde o sucessor de um elemento ocupa uma posição física subsequente.

- Exemplo: Arrays

Listas com Representação Sequencial

- Inserção de um elemento na posição **i**.
  - Causa o deslocamento a direita de todos os elementos da Lista.

- Remoção de um elemento na posição **i**,
  - Requer o deslocamento à esquerda dos elementos E_{i+1} até o último elemento da Lista.



Listas com Representação Sequencial

- **Vantagens:**
  - Acesso ao i-ésimo elemento é imediato;
  - Algoritmos simples.

- **Desvantagens:**
  - Não usa memória de forma eficiente
    - Aloca um espaço finito e predeterminado;
  - Intensa movimentação na inserção/remoção de elementos;

Listas com Representação Sequencial

- Quando usar:
  - Listas pequenas;
  - Inserção/Remoção no fim da Lista;
  - Tamanho máximo bem definido.

Implementação de Listas Sequenciais

- Operações Básicas
  - Criação da lista vazia;
  - Verificar se a lista está vazia;
  - Verificar se a lista está cheia;
  - Obter o tamanho da lista;
  - Obter/modificar o valor do elemento de uma determinada posição na lista;
  - Inserir um elemento em uma determinada posição;
  - Retirar um elemento de uma determinada posição.

Implementação de Listas Sequenciais

```
public class ListaSeq {

   // Vetor que contém os dados da lista
   private int dados[];
   private int tamAtual;
   private int tamMax;


   public ListaSeq(){
       tamMax = 100;
       tamAtual = 0;
       dados = new int[tamMax];
   }
```

Estrutura do tipo

Implementação de Listas Sequenciais

```
// Definição das Operações

/** Verifica se a Lista está vazia */
public boolean vazia(){
   if (tamAtual == 0 ) return true;
   else  return false;
}

/**Verifica se a Lista está cheia */
public boolean cheia(){
   if (tamAtual == tamMax) return true;
   else     return false;
}

//continua...
```

Operações

Implementação de Listas Sequenciais

```
/**Obtém o tamanho da Lista*/
public int tamanho(){
      return tamAtual;
}

// continua...
```

Implementação de Listas Sequenciais

```
/** Obtém o i-ésimo elemento de uma lista.
    Retorna -1 se a posição for inválida. */
public int elemento(int pos){
      int dado;
      /* Se posição estiver fora dos limites
         <= 0 ou > tamanho da lista */
      if ((pos > tamAtual) || (pos <= 0))
            return -1;

       dado = dados[pos-1];
       return dado;
}

//continua...
```

Implementação de Listas Sequenciais

```
/** Retorna a posição de um elemento pesquisado.
    Retorna -1 caso não seja encontrado */
public int posicao (int dado){
      /* Procura elemento a elemento.
      Se estiver, retorna posição no array+1 */
      for (int i = 0; i < tamAtual; i++){
            if (dados[i] == dado){
                  return (i + 1);
            }
      }
      return -1;
}
// continua...
```

Implementação de Listas Sequenciais

```
/**Insere um elemento em uma determinada posição.
   Retorna true se a insercao funcionar e
   false caso contrário. */
public boolean insere (int pos, int dado){
   if (cheia() || (pos > tamAtual+1) || (pos<=0)){
       return false;
   }

   for (int i = tamAtual; i >= pos; i--){
       dados[i] = dados[i-1];
   }

   dados[pos - 1] = dado;
   tamAtual++;
   return true;
}
```

Implementação de Listas Sequenciais

```
/** Remove um elemento de uma determinada posição
    Retorna o valor do elemento removido e
   -1 caso a remoção falhe  */
public int remove(int pos){
   int dado;
   if ((pos > tamAtual) || (pos < 1 ))
       return -1;

   dado = dados[pos-1];
   for (int i = pos - 1; i < tamAtual - 1; i++){
       dados[i] = dados[i+1];
   }
   tamAtual--;
   return dado;
}
}
```

Alocação Dinâmica de Memória

Alocação Dinâmica de Memória

- Feita de acordo com a demanda apresentada durante a execução do programa
  - Pode aumentar ou diminuir durante a execução do programa;

- Usada quando a quantidade de memória necessária não pode ser determinada a priori

- Tipos de Dados Dinâmicos
  - São tipos de dados cujo tamanho pode aumentar ou diminuir durante a execução do programa.
  - Ex: **Listas Encadeadas.**

Alocação Dinâmica de Memória em C

- Feita por meio de ponteiros e funções da biblioteca padrão
- Incluir: `<stdlib.h>`

FUNÇÃO	DESCRIÇÃO RESUMIDA
malloc()	Aloca um dado número especificado de bytes em memória e retorna um ponteiro para o início do bloco de memória alocado
calloc()	Similar a **malloc()**, mas inicia todos os bytes alocados com zeros e permite a alocação de memória de mais de um bloco numa mesma chamada
realloc()	Modifica o tamanho de um bloco previamente alocado
free()	Libera o espaço de um bloco de memória alocado com **malloc()**, **calloc()** ou **realloc()**

Alocação Dinâmica de Memória em Java e C++

- **Alocação**: Feita por meio do operador `new`

Em Java

```
Passageiro p;
...
p = new Passageiro();
```

Em C++

```
Passageiro *p;
...
p = new Passageiro();
```

- **Liberação**: Feita por meio do operador `delete` (C++)

Em Java

```
p = null;
```

Em C++

```
delete p;
```

Alocação Dinâmica de Memória em Java e C++

- **Alocação**: Feita por meio do operador `new`

Em Java

```
Passageiro p;
...
p = new Passageiro();
```

Em C++

```
Passageiro *p;
...
p = new Passageiro();
```

- **Liberação**:

Em Java

```
p = null;
```

Na realidade, em Java, a liberação de memória é feita automaticamente pelo Garbage Collection. Essa operação apenas sugere ao Garbage Collection que a região não é mais usada

Listas Simplesmente Encadeadas

Listas Encadeadas

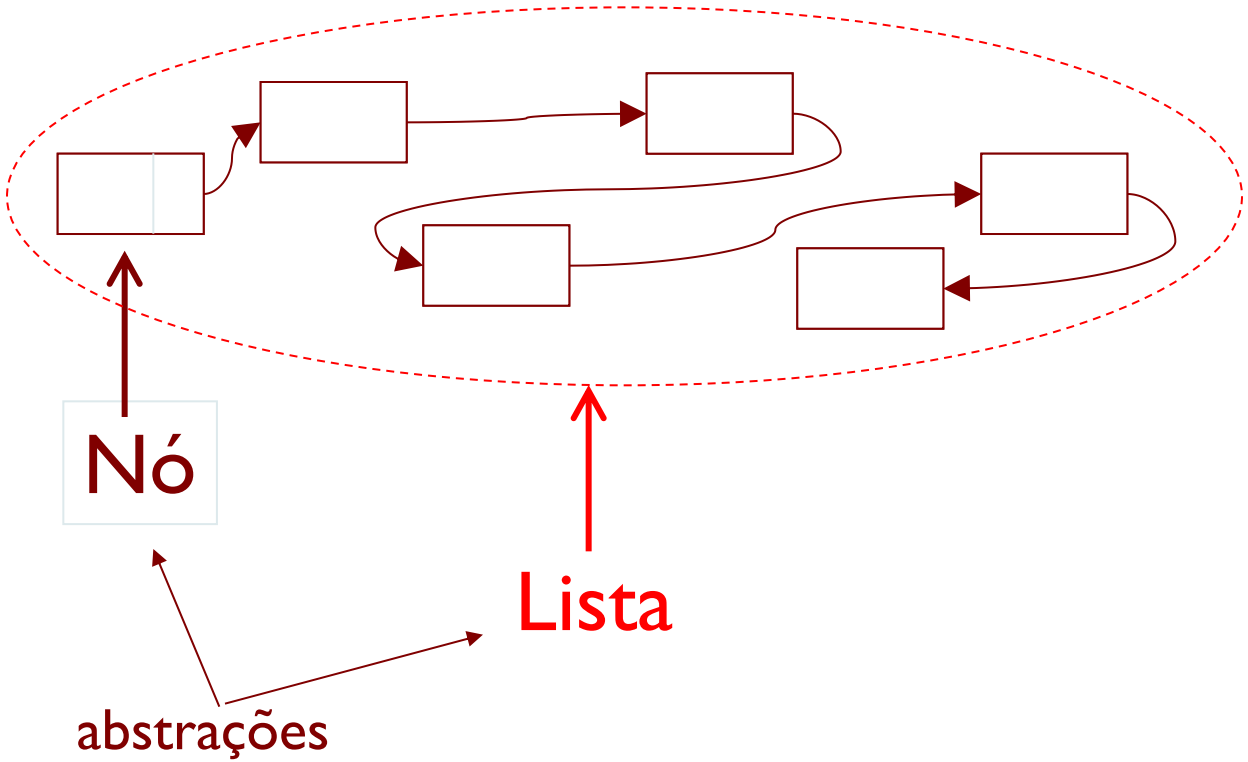
- São **estrutura de dados lineares** e **dinâmicas**.

- **Nº de elementos (nós) da lista** pode **aumentar ou diminuir** dinamicamente à medida que novos elementos são **inseridos ou removidos**

- Normalmente, inicia vazia e depois os elementos vão sendo inseridos ou removidos um a um.

Listas Simplesmente Encadeadas

- Criando uma abstração para uma LE



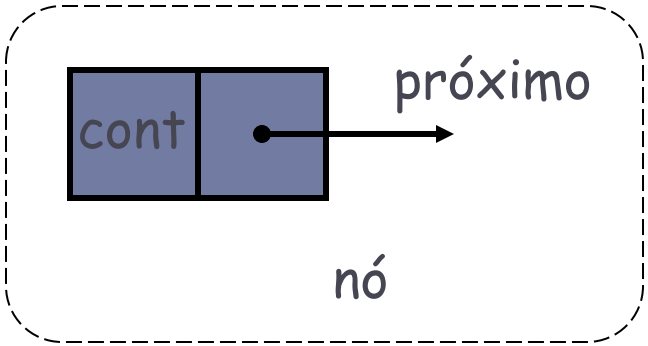
Listas Encadeadas

- Vantagens:
  - Melhor aproveitamento da memória;
  - Menor overhead para inserção/remoção na lista
    - Não há necessidade de deslocamentos de nós

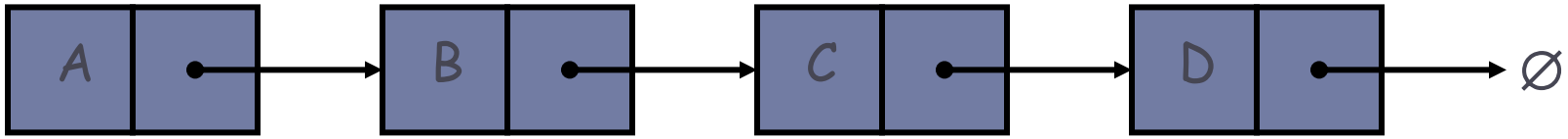
- Desvantagens:
  - Algoritmos mais complexos;
  - Uso de apontadores;
  - O acesso aos nós deve ser feito de forma sequencial.

Lista Simplesmente Encadeada

- É uma **estrutura de dados** que consiste de uma **sequência de nós**

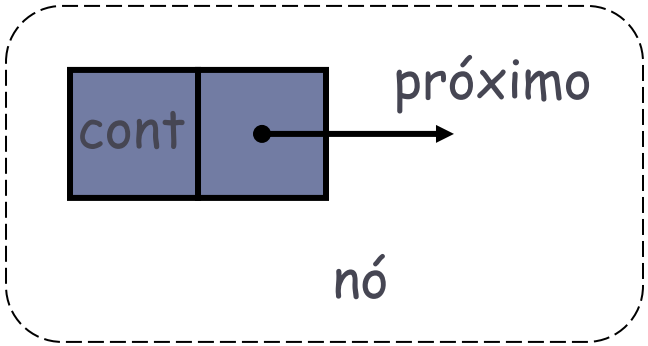


- Cada **nó** armazena:
  - O **conteúdo** do elemento
  - Uma **ligação para o próximo nó**



O Tipo (Classe) "Nó"

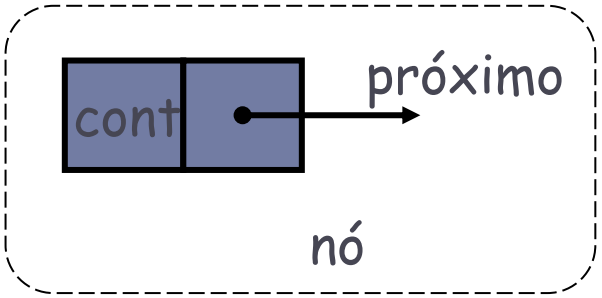
- Possui os campos de informação
- Possui um campo de ligação com o próximo elemento do tipo Nó
- As operações sobre nó são:
  - Atualiza informação
  - Atualiza próximo
  - Recupera informação
  - Recupera próximo



Implementação do tipo "Nó"

- Em Java:

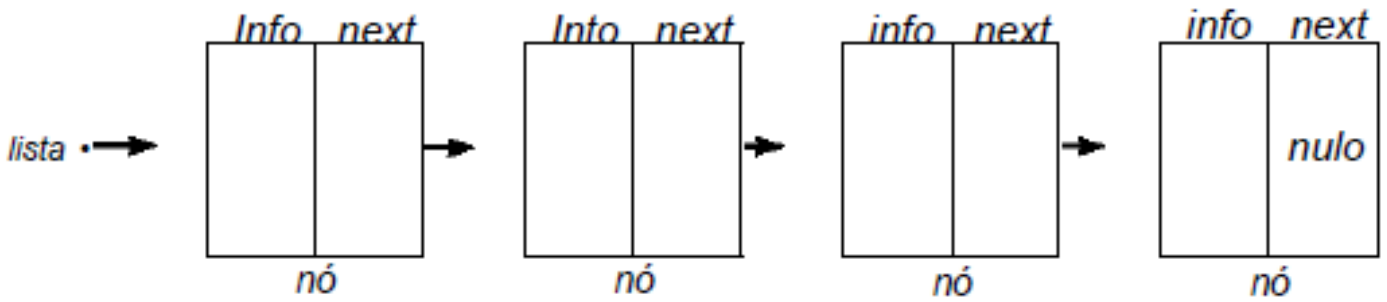
```
public class No{
   private int conteudo;
   private No proximo;

   public No(){
      prox = null;
   }

   // Métodos get e set
}
```



O Tipo (Classe) "Lista Encadeada"

- Possui um campo que referencia o **início da lista**
  - Também chamado de **cabeça da lista (head)**

- Possui um campo que representa o nº total de nós da lista.



Implementação do tipo "Lista Encadeada"

- Em Java:

```
public class Lista{
    private No cabeca;
    private int tamanho;

    public Lista(){
        cabeca = null;
        tamanho = 0;
    }
}
```

Implementação de Listas Encadeadas

- Operações Básicas
  - Criação da lista vazia;
  - Verificar se a lista está vazia;
  - Verificar se a lista está cheia;
  - Obter o tamanho da lista;
  - Obter/modificar o valor do elemento de uma determinada posição na lista;
  - Inserir um elemento em uma determinada posição;
  - Retirar um elemento de uma determinada posição.

Implementação de Listas Sequenciais

```
// Definição das Operações

/** Verifica se a Lista está vazia */
public boolean vazia(){
   if (tamAtual == 0 ) return true;
   else  return false;
}


//continua...
```

Implementação de Listas Encadeadas

```
/**Obtém o tamanho da Lista*/
public int tamanho() {
   return tamanho;
}

// ou
public int tamanho() {
   No p = cabeca;
   int cont = 0;
   while(p != null){
       p = p.getProx();
       cont++;
   }
   return cont;
}
```

Implementação de Listas Encadeadas

```
/** Obtém o i-ésimo elemento de uma lista
    Retorna o valor encontrado. */
public int elemento (int pos) {
      No aux = cabeca;
      int cont = 1;
      if (vazia()) return -1; // Consulta falhou

      if ((pos < 1) || (pos > tamanho))
            return -1; // Posicao invalida

      // Percorre a lista do 1o elemento até pos
      while (cont < pos){
            aux = aux.getProx();
            cont++;
      }
      return aux.getConteudo();
}
```

Implementação de Listas Encadeadas

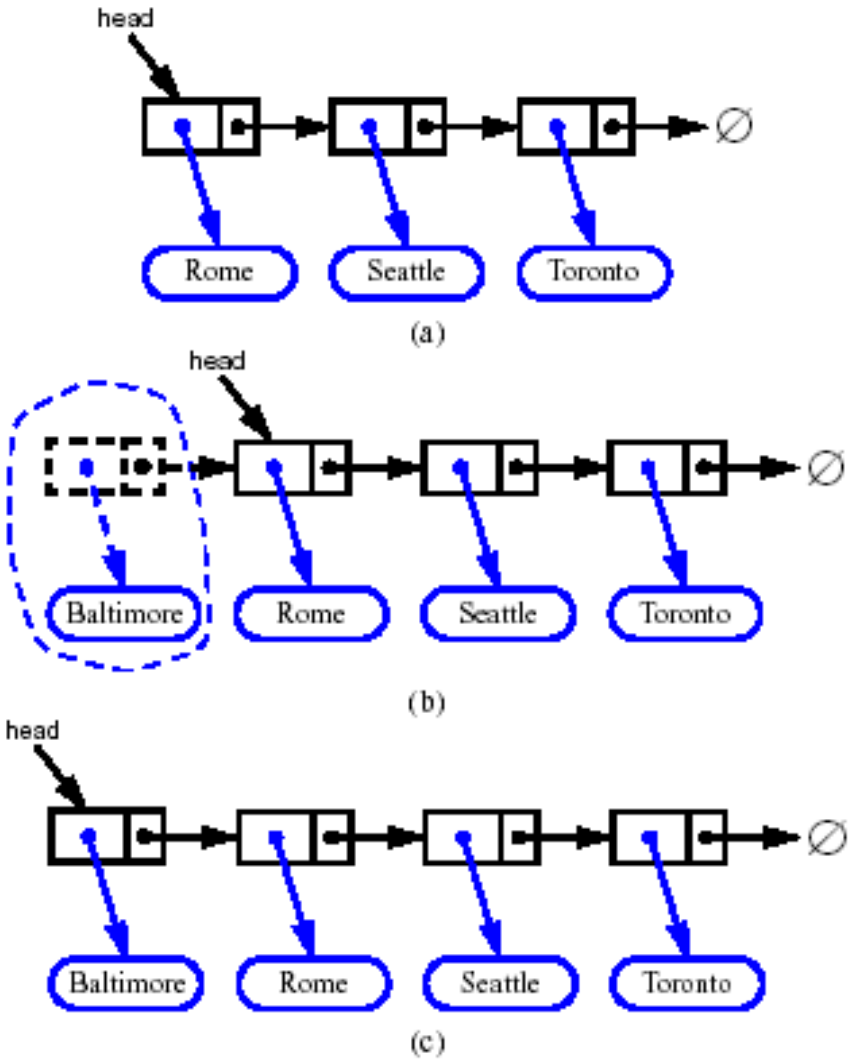
```
/**Retorna a posição de um elemento pesquisado.
   Retorna -1 caso não seja encontrado */
public int posicao (int dado) {
      int cont = 1;
      No aux;

      if (vazia()) return 0;
      aux = cabeca;
      while (aux != null) {
         if (aux.getConteudo() == dado)
            return cont;
         aux = aux.getProx();
         cont++;
      }
      return -1;
}
```

Implementação de Listas Encadeadas

```
/**Insere um elemento em uma determinada posição
   Retorna true se conseguir inserir e
   false caso contrario */
boolean insere(int pos, int dado) {
   if ((vazia()) && (pos != 1)) return false;

   if (pos == 1){ // insercao no inicio da lista
       return insereInicioLista(dado);
   }
   else if (pos == tamanho+1){ // inserção no fim
       return insereFimLista(dado);
   }
   else{  // inserção no meio da lista
       return insereMeioLista(pos, dado);
   }
}
```

Inserção de nó no início da Lista

1. Aloque um novo nó
2. Faça o campo próximo do novo nó apontar para o nó cabeça da lista
3. Atualize o campo que aponta para a cabeça para apontar para o novo nó
4. Incremente o contador de nós

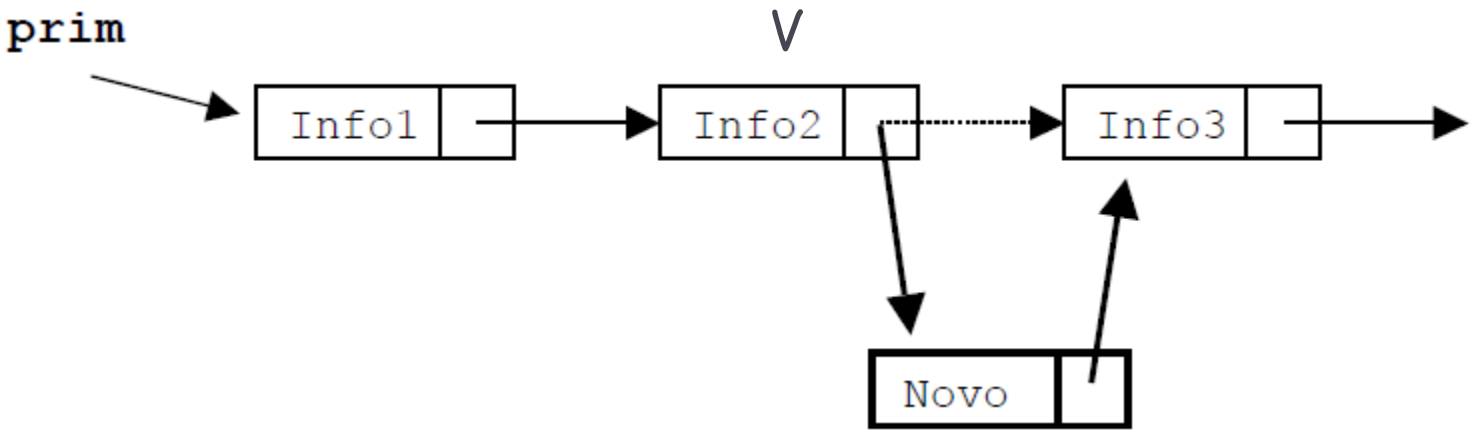


Inserção de nó no início da lista

```
/** Insere nó em lista vazia */
private boolean insereInicioLista(int valor) {
      // Aloca memoria para um novo no */
      No novoNo = new No();

      // Insere novo elemento na cabeca da lista
      novoNo.setConteudo(valor);
      novoNo.setProx(cabeca);
      cabeca = novoNo;
      tamanho++;
      return true;
}
```

Inserção de nó no meio da lista

1. Use uma variável auxiliar do tipo Nó para localizar o nó "V" após o qual se deseja inserir o novo nó
2. Aloque um novo nó
3. Faça o campo próximo do novo nó apontar para o nó apontado pelo campo próximo do nó "V"
4. Faça o campo próximo do nó "V" apontar para o novo nó

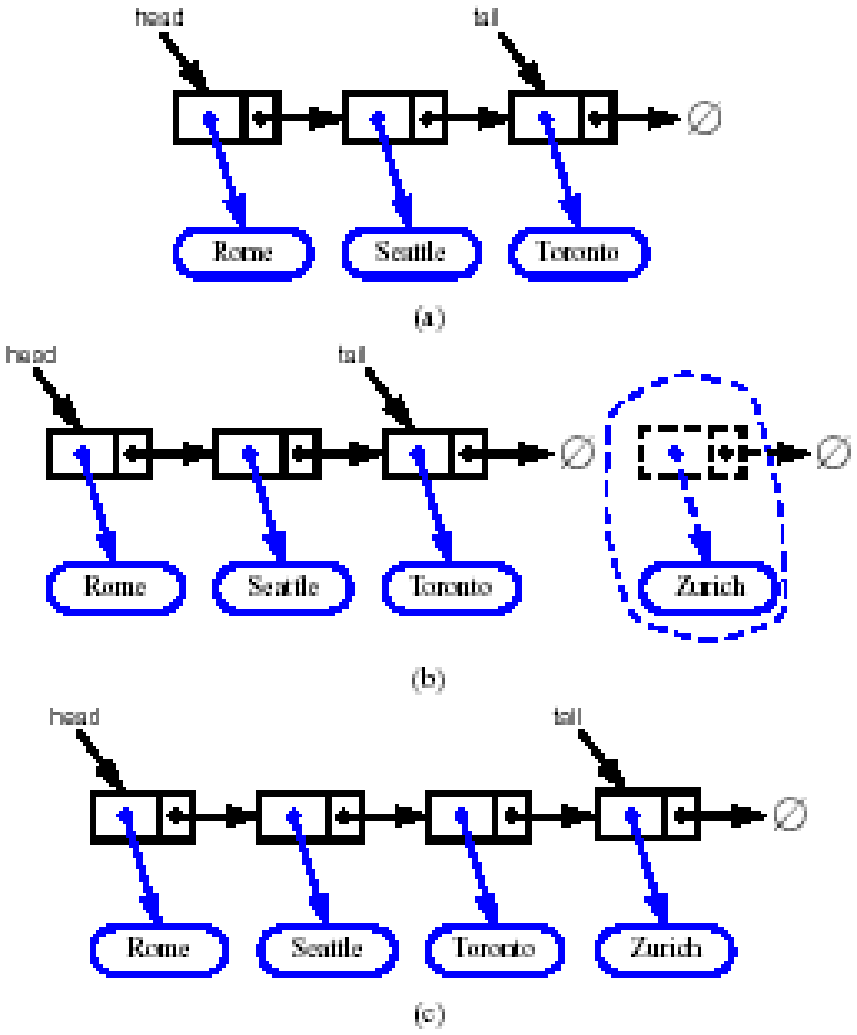


Inserção de nó no meio da lista

```
/** Insere nó no meio da lista */
private boolean insereMeioLista(int pos, int dado){
   int cont = 1;
   No novoNo = new No(); // Aloca memoria para novo no
   novoNo.setConteudo(dado);

   // Localiza a pos. onde será inserido o novo nó
   No aux = cabeca;
   while ((cont < pos-1) && (aux != null)){
       aux = aux.getProx();
       cont++;
   }
   if (aux == null) return false;

   novoNo.setProx(aux.getProx());
   aux.setProx(novoNo);
   tamanho++;
   return true;
}
```

Inserção de nó no fim da lista

1. Localize a cauda da lista
2. Aloque um novo nó
3. Faça o campo próximo do novo nó apontar para null
4. Faça o campo próximo do nó cauda apontar para o novo nó
5. Incremente o contador de nós



Inserir o nó no fim da lista em C

```
/** Insere nó no fim da lista */
 private boolean insereFimLista(int dado){
      No novoNo = new No();
      novoNo.setConteudo(dado);

      // Procura o final da lista
      No aux = cabeca;
      while(aux.getProx() != null){
            aux = aux.getProx();
      }

      novoNo.setProx(null);
      aux.setProx(novoNo);
      this.tamanho++;
      return true;
}
```

Implementação de Listas Encadeadas

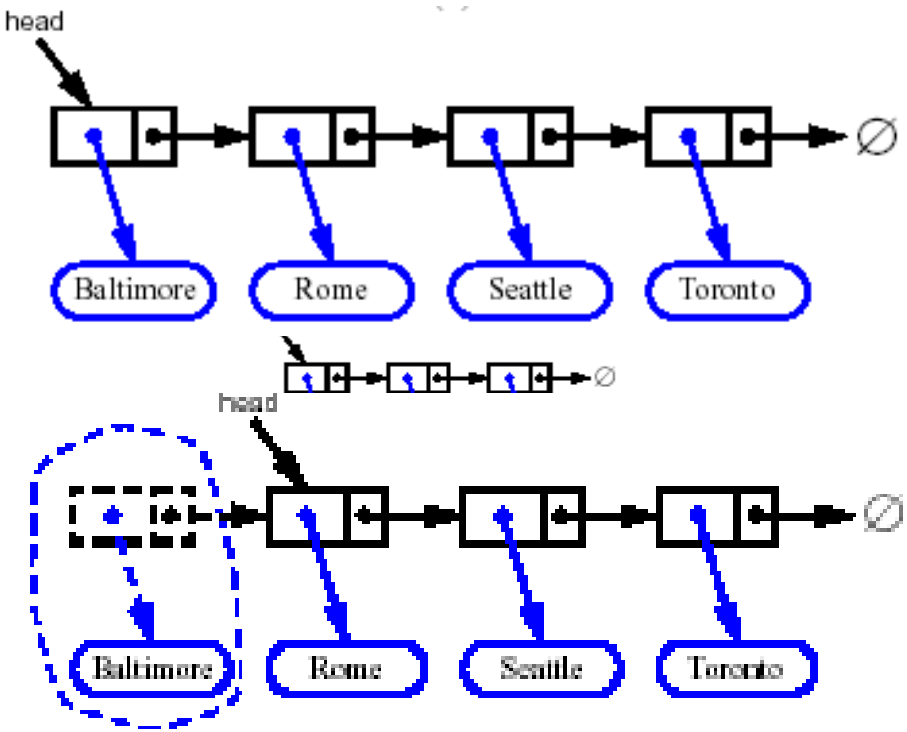
```
/**Remove um elemento de uma determinada posição
   Retorna o valor a ser removido.
   -1 se a posição for inválida ou a lista vazia
*/public int remove(int pos) {
   if (vazia()) return -1; // Lista vazia

   // remoção do elemento da cabeça da lista
   if (pos == 1){
       return removeInicioLista();
   }
   // remoção em outro lugar da lista
   else{
       return removeNaLista(pos);
   }
}
```

Remover um nó da cabeça da lista

1. Use uma variável auxiliar do tipo Nó para apontar para a cabeça da lista

2. Atualize o campo que aponta para a cabeça da lista para apontar para o próximo nó na lista

3. Libera a memória do nó removido
   - Dispensável em Java

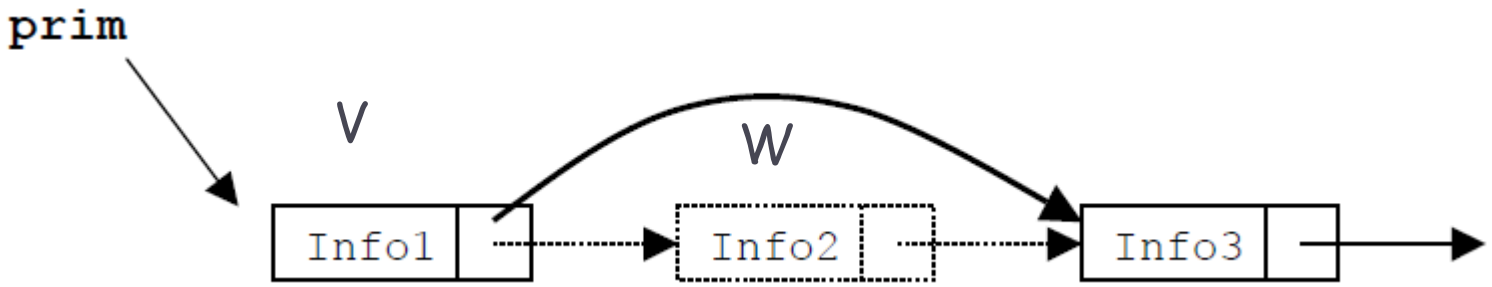


Implementação de Listas Encadeadas

```
/** Remove elemento do início da lista */
 private int removeInicioLista(){
   No p = cabeca;
   int dado = p.getConteudo();

   // Retira o 1o elemento da lista (p)
   cabeca = p.getProx();
   tamanho--;

   // Sugere ao garbage collector que libere a
   // memoria da regiao apontada por p
   p = null;

   return dado;
}
```

Remover um nó do meio da lista

1. Use uma variável auxiliar do tipo Nó para localizar o nó anterior "V" ao nó a ser removido da lista
2. Use uma outra variável auxiliar do tipo Nó para apontar para o nó "W" a ser removido da lista
3. Faça o campo próximo do nó "V" apontar para o nó apontado pelo campo próximo do nó a ser removido da lista
4. Libere a memória do nó removido



Implementação de Listas Encadeadas

```
/** Remove elemento no meio da lista */
private int removeNaLista(int pos){
   No atual = null, antecessor = null;
   int dado = -1, cont = 1;

   atual = cabeca;
   while((cont < pos) && (atual != null)){
       antecessor = atual;
       atual = atual.getProx();
       cont++;
   }

   if (atual == null)
       return -1;

// continua...
```

Implementação de Listas Encadeadas

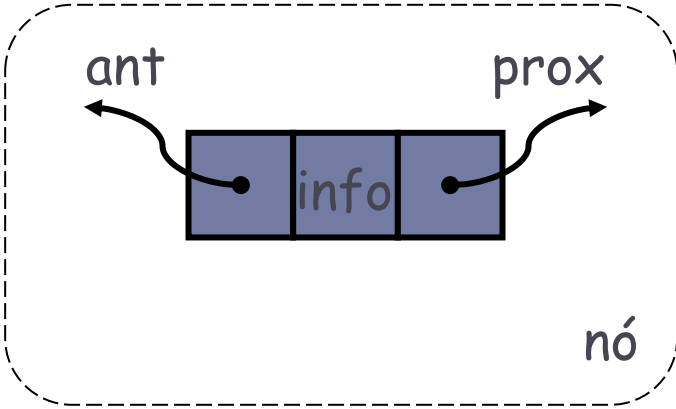
```
    // retira o elemento da lista
    dado = atual.getConteudo();
    antecessor.setProx(atual.getProx());
    tamanho--;

    // sugere ao garbage collector que libere a memoria
    // da regiao apontada por p
    atual = null;
    return dado;
}
```

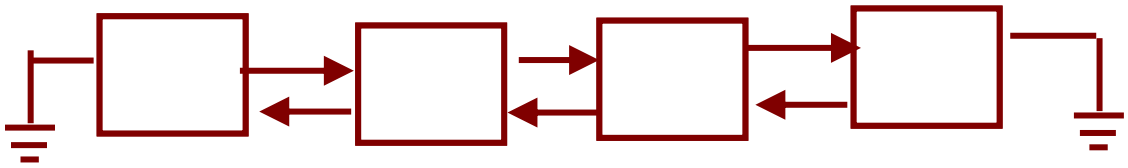
Listas Duplamente Encadeadas

Listas Duplamente Encadeadas

- Listas que permitem a movimentação nos dois sentidos.

- Nós possuem **duas ligações (ponteiros)**:
  - Uma ligação para o **próximo nó**;
  - Uma ligação para o **nó anterior**;

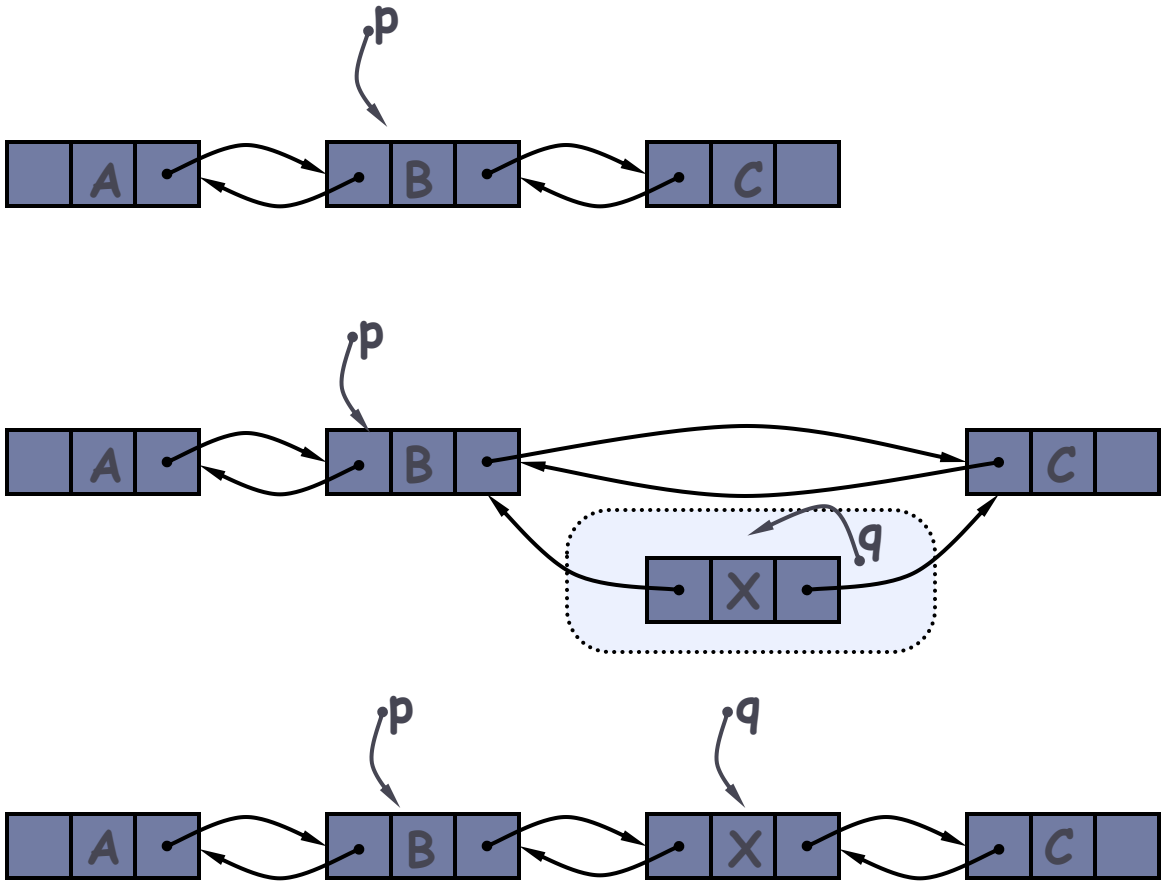




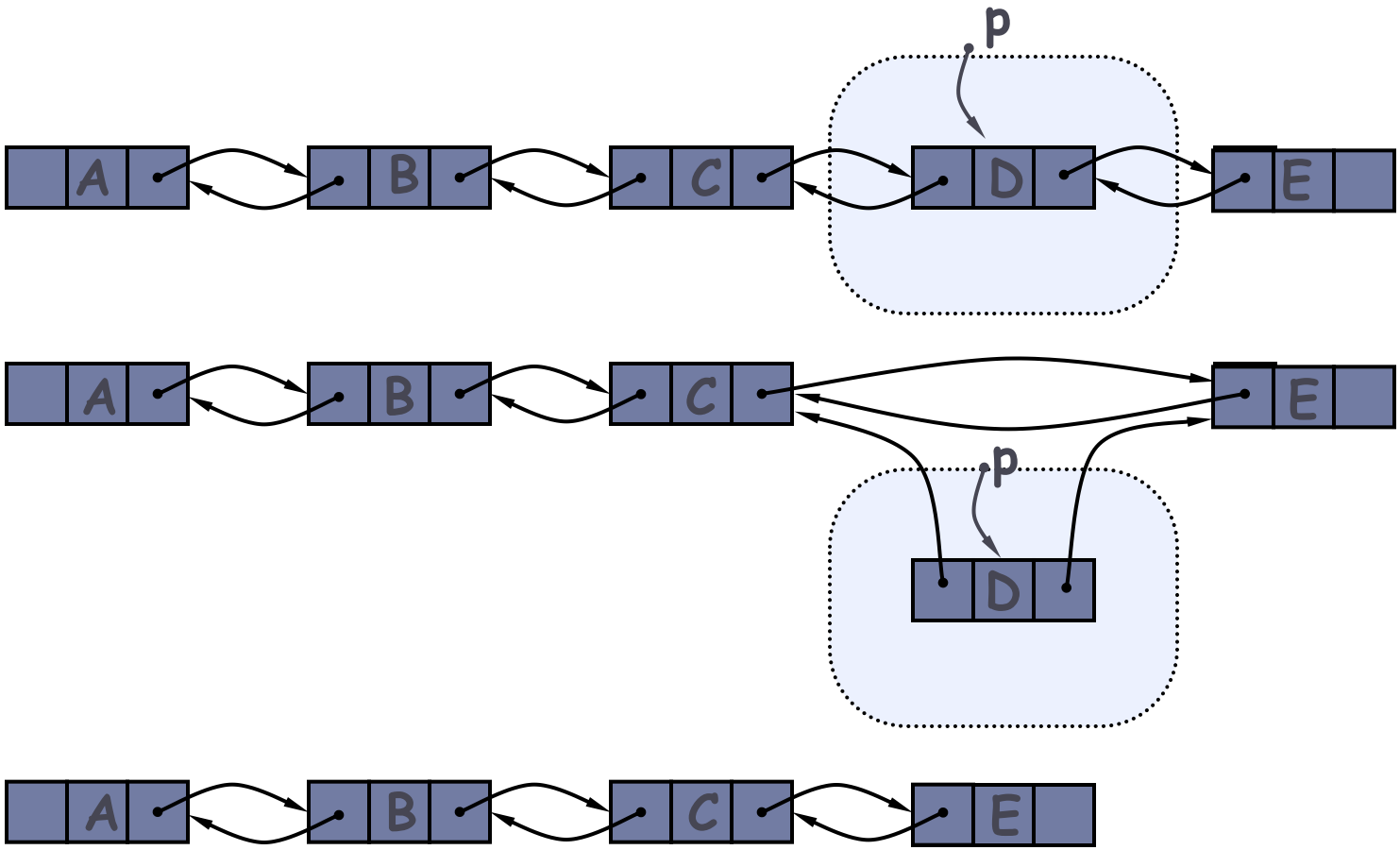
Listas Duplamente Encadeadas

- Características:
  - Ocupam mais memória do que uma lista simplesmente encadeada.
  - Possibilidade de se movimentar nos dois sentidos simplifica a implementação de algumas funções para listas

Inserir novo elemento na lista



Remover elementos na LDE



Implementação de Listas Duplamente Encadeadas

- Operações Básicas
  - Criação da lista vazia;
  - Verificar se a lista está vazia;
  - Obter o tamanho da lista;
  - Obter/modificar o valor do elemento de uma determinada posição na lista;
  - Inserir um elemento em uma determinada posição;
  - Retirar um elemento de uma determinada posição.

Implementação de Listas Duplamente Encadeadas (LDEs)

```
public class No{
    private No ant;
    private int conteudo;
    private No prox;

    public No(){
        ant = null;
        prox = null;
    }

    // Métodos get e set
}
```

```
public class Lista{
    private No cabeca;
    private No cauda;
    private int nElementos;

    public Lista(){
        cabeca = null;
        cauda = null;
        nElementos = 0;
    }
}
```

Implementação de LDE

```
// Definição das Operações

/** Verifica se a Lista está vazia */
public boolean vazia(){
   if (tamAtual == 0 ) return true;
   else  return false;
}


//continua...
```

Implementação de LDE

```
/**Obtém o tamanho da Lista*/
public int tamanho() {
   return tamanho;
}
```

Implementação de LDE

```
/** Obtém o i-ésimo elemento de uma lista
    Retorna o valor encontrado. */
public int elemento (int pos) {
      No aux = cabeca;
      int cont = 1;
      if (vazia()) return -1; // Consulta falhou

      if ((pos < 1) || (pos > tamanho))
            return -1; // Posicao invalida

      // Percorre a lista do 1o elemento até pos
      while (cont < pos){
            aux = aux.getProx();
            cont++;
      }
      return aux.getConteudo();
}
```

Implementação de LDE

```
/**Retorna a posição de um elemento pesquisado.
   Retorna -1 caso não seja encontrado */
public int posicao (int dado) {
      int cont = 1;
      No aux;

      if (vazia()) return 0;
      aux = cabeca;
      while (aux != null) {
         if (aux.getConteudo() == dado)
            return cont;
         aux = aux.getProx();
         cont++;
      }
      return -1;
}
```

Inserção em LDEs

```
/**Insere um elemento em uma determinada posição
   Retorna true se conseguir inserir e
   false caso contrario */
boolean insere(int pos, int dado) {
   if ((vazia()) && (pos != 1)) return false;

   if (pos == 1){ // insercao no inicio da lista
      return insereInicioLista(dado);
   }
   else if (pos == tamanho+1){ // inserção no fim
      return insereFimLista(dado);
   }
   else{  // inserção no meio da lista
      return insereMeioLista(pos, dado);
   }
}
// continua...
```

Inserção em LDEs

```
/** Insere nó em lista vazia */
private boolean insereInicioLista(int valor) {
      // Aloca memoria para um novo no */
      No novoNo = new No();

      // Insere novo elemento na cabeca da lista
      novoNo.setConteudo(valor);
      novoNo.setProx(inicio);

      novoNo.setAnt(null);
      if (vazia()) fim = novoNo;
      else inicio.setAnt(novoNo);

      inicio = novoNo;
      tamanho++;
      return true;
}
```

Inserção em LDEs

```
/** Insere nó no meio da lista */
private boolean insereMeioLista(int pos, int dado){
     int cont = 1;
     No novoNo = new No();
     novoNo.setConteudo(dado);

     No aux = inicio;
     while ((cont < pos-1) && (aux != null)){
       aux = aux.getProx(); cont++;
     }

     if (aux == null) { return false; }
     novoNo.setAnt(aux); // Nova instrucao
     novoNo.setProx(aux.getProx());
     aux.getProx().setAnt(novoNo); // Nova instrucao
     aux.setProx(novoNo);
    tamanho++;
     return true;
}
```

Inserção em LDEs

```
/** Insere nó no fim da lista */
 private boolean insereFimLista(int dado){
      No novoNo = new No();
      novoNo.setConteudo(dado);

      // Procura o final da lista
      No aux = fim;

      novoNo.setProx(null);
      aux.setProx(novoNo);
      novoNo.setAnt(fim);
      fim.setProx(novoNo);
      fim = novoNo;
      this.tamanho++;
      return true;
}
```

Remoção em LDEs

```
// Remove um elemento de uma determinada posição
private int remove(int pos) {
    int  ret;
    if (vazia()) {return (0);} // lista vazia
    //remoção do elemento de uma lista unitária
    if ((pos == 1) && (tamanho() == 1)){
       return removeInicioListaUnitaria();
    }
    //remoção do elemento da cabeça da lista
    else if (pos == 1){
       return removeInicioLista();
    }
// continua...
```

Remoção em LDEs

```
// continua...
    // remoção no final da lista
    else if (pos == tamanho()){
        return removeFimLista();
    }
    else{ // remoção no meio da lista
        return removeMeioLista(pos);
    }
}
```

Remoção em LDEs

```
// Remove elemento do início da lista
private int removeInicioListaUnitaria(){
    int dado = inicio.getConteudo();
    inicio = null;
    fim = null;
    tamanho--;
    return dado;
}
```

Implementação de Listas Encadeadas

```
/** Remove elemento do início da lista */
 private int removeInicioLista(){
   No p = cabeca;
   int dado = p.getConteudo();

   // Retira o 1o elemento da lista (p)
   cabeca = p.getProx();
   p.getProx().setAnt(null);
   tamanho--;

   // Sugere ao garbage collector que libere a
   // memoria da regiao apontada por p
   p = null;

   return dado;
}
```

Implementação de LDEs

```
// Remove elemento do fim da lista
private int removeFimLista(){
   No p = fim;
   int dado = p.getConteudo();

   fim.getAnt().setProx(null);
   fim = fim.getAnt();
   tamanho--;

   p = null;
   return dado;
}
```

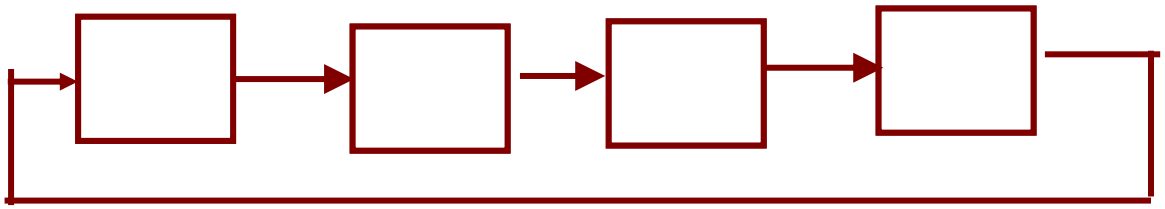
Implementação de LDEs

```
// Remove elemento do início da lista
private int removeMeioLista(int pos){
   No p = inicio;
   int n = 1;

   while((n <= pos-1) && (p != null)){
       p = p.getProx(); n++;
   }

   if (p == null) return -1; // pos. inválida
   int dado = p.getConteudo();
   p.getAnt().setProx(p.getProx());
   p.getProx().setAnt(p.getAnt());
   tamanho--;
   p = null;
   return dado;
}
```

Listas Circulares

Listas Encadedas Circulares

- Listas onde o **último nó** aponta para o **1º elemento**.
- Elas podem ser **simples ou duplamente encadeada.**
- Útil quando:
  - Quando se deseja acessar os elementos da lista um por vez em *loop*
  - Ex: Sistema Operacional compartilha o tempo da CPU entre processo com uma lista circular.



Universidade Federal da Paraíba

Centro de Informática

Departamento de Informática

Estrutura de Dados
Listas

- Tiago Maritan
- tiago@ci.ufpb.br