Listas Lineares

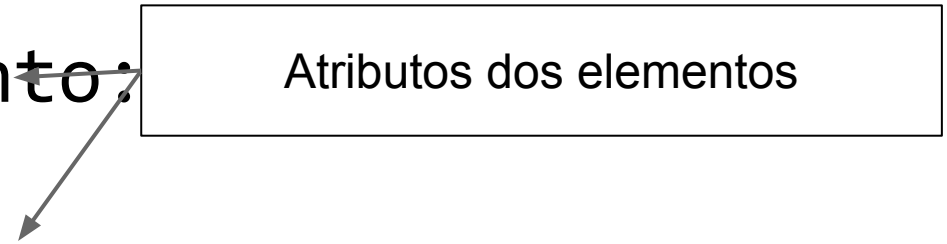
- Consideremos o problema *de* agrupar diversos elementos que, de alguma forma, relacionam-se entre si
  - Admitiremos que cada elemento possui um identificador distinto chamado de ***chave*** e possivelmente outros campos

Ex.: agrupamento de Alunos (chave: matrícula), agrupamento de Logs de Acesso a Sistemas (chave: data/hora e usuário), agrupamento de números (chave: o próprio número!), etc.

Listas Lineares

- Implementação de um elemento não-escalar:

```
estrutura Elemento:
   Campo1: Tipo1
   ...
   CampoK: TipoK
```



Listas Lineares

- Exemplo:

```
estrutura Aluno:
    Matricula: Inteiro
    Nome: Caracter[50]
    DataNasc: DataHora
    Endereco: Caracter[500]
    Formado: Lógico
    CR: Real
```

Listas Lineares

- Existem diversas maneiras ***como*** um agrupamento de elementos pode ser implementado — discutiremos ao longo do curso os prós e contras de cada implementação

- Cada forma de implementação visa fornecer **um conjunto de operações** que se pode realizar sobre o agrupamento de forma **mais eficiente** que as demais implementações — **e não há uma forma mais eficiente para todo possível conjunto de operações**

Listas Lineares

- ***Listas lineares*** é a estrutura de dados que armazena os elementos em lista (onde há o conceito de primeiro elemento, segundo elemento, terceiro elemento, etc.) e provê o seguinte conjunto de operações:
  - Inserção de um elemento
  - Remoção de um elemento
  - Busca de um elemento

ListaLinear <TChave, TElem>

procedimento Enumera(**ref** L: ListaLinear)

Enumera (lista) a chave de cada elemento de L

função Busca(**ref** L: ListaLinear, c: <TChave>): <TElem>

Obtém o elemento de L com chave c ou "NULO" se inexistente

função BuscaEm(**ref** L: ListaLinear, Pos: Inteiro): <TElem>

Obtém o elemento de L com posição Pos

procedimento Insere(**ref** L: ListaLinear, c: <TChave>, x: <TElem>)

Insere x com chave c em L

procedimento InsereEm(**ref** L: ListaLinear, Pos: Inteiro,
c: <TChave>, x: <TElem>)

Insere x em L na posição Pos (deslocando elementos a partir da posição Pos à direita)

função Remove(**ref** L: ListaLinear, c: <TChave>): <TElem>

Remove e retorna o elemento de L com chave c

função RemoveEm(**ref** L: ListaLinear, Pos: Inteiro): <TElem>

Remove o elemento na posição Pos de L (deslocando elementos a partir da posição Pos+1 à esquerda)

função Tamanho(**ref** L: ListaLinear): Inteiro

Obtém o número de elementos em L

Listas Lineares

- Exemplo 1:

```
var L: ListaLinear <Inteiro, Inteiro>

Insere(L, 1, 1)
Insere(L, 2, 2)
Enumera(L) //saída: 1,2 ou 2,1
Remove(L, 1)
Enumera(L) //saída: 2
```

Listas Lineares

- Exemplo 2:

```
var L: ListaLinear <Inteiro, Aluno>
var A1, A2, A: Aluno
A1.Matricula, A1.Nome[1] ← 1, 'A'
A2.Matricula, A2.Nome[1] ← 2, 'B'
Insere(L, A1.Matricula, A1)
Insere(L, A2.Matricula, A2)
A ← Busca(L, 2)
escrever (A.Nome[1]) //saída: B
A ← Busca(L, 1)
escrever (A.Nome[1]) //saída: A
A1.Nome[1] ← 'C'
A ← Busca(L, 1)
escrever (A.Nome[1]) //saída: A
...
```

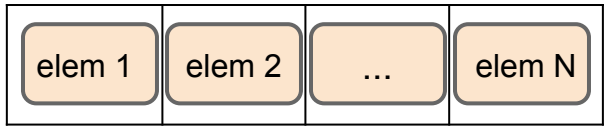
Listas Lineares

- Exemplo 3:

```
var L: ListaLinear <Inteiro, ^Aluno>
var A1, A2, A: ^Aluno
alocar(A1)
alocar(A2)
A1^.Matricula, A1^.Nome[1] ← 1, 'A'
A2^.Matricula, A2^.Nome[1] ← 2, 'B'
Insere(L, A1^.Matricula, A1)
Insere(L, A2^.Matricula, A2)
A ← Busca(L, 2)
escrever (A^.Nome[1]) //saída: B
A ← Busca(L, 1)
escrever (A^.Nome[1]) //saída: A
A1^.Nome[1] ← 'C'
A ← Busca(L, 1)
escrever (A^.Nome[1]) //saída: C
...
```

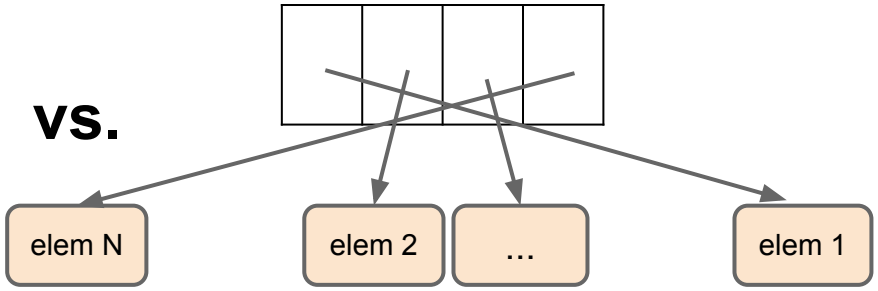
Listas Lineares

- Qual o critério para se usar uma lista de de ponteiros para elementos ao invés de uma lista para elementos?



lista de elementos

vs.



lista de ponteiros para elementos

Listas Lineares

- Quando os elementos se movem entre as posições (exemplo, uso frequente de ordenações), o uso de ponteiros evita transferências de dados em memória

- Elementos que participam em mais de uma lista linear podem manter seus dados em um único lugar com o uso de ponteiros, ao invés de criar duplicações de dados em memória

- Quando a quantidade de elementos é relativamente alta, devido a organização do sistema operacional, **é necessário** o uso de **alocação dinâmica** (comando **alocar**) para evitar o estouro de pilha

Listas Lineares

- Implementação de Listas Lineares depende da alocação em memória dos elementos:
  - **Alocação Sequencial**: elementos ocupam memória em posições contíguas. Acesso a elementos em tempo constante, pois a posição de cada elemento pode ser inferida
  - **Alocação Encadeada**: elementos não necessariamente ocupam posições contíguas em memória. Elementos devem ser consultados a partir das posições de elementos já conhecidos, avançando-se para os próximos elementos (cada elemento armazena um ponteiro para o próximo elemento na sequência)

Listas Lineares

- Implementação de Listas Lineares varia também dependendo da ordem das chaves:
  - **Não-Ordenada**: elementos são guardados em ordem arbitrária, não tendo relação com os valores das respectivas chaves
  - **Ordenada**: elementos são guardados em ordem de chave (caso o contrário não seja dito, ascendentemente por convenção)

Listas Lineares

- Qual tipo de lista é o melhor? A resposta é "DEPENDE!". O uso dos elementos (ou seja, as operações sobre a lista e em que freqüência elas ocorrem) é que determinará qual é a melhor implementação de lista linear para o problema em questão

Alocação Sequencial (elementos armazenados num vetor)

Listas Lineares

```
var MAX_N: Inteiro ← <NÚMERO MÁXIMO DE ELEMENTOS>

estrutura No <TChave, TElem>:
    Chave: <TChave>
    Elem: <TElem>

estrutura ListaLinear <TChave, TElem>:
    Val[1..MAX_N]: No <TChave, TElem>
    N: Inteiro //número corrente de elementos na lista

procedimento Constroi(ref L: ListaLinear)
    L.N ← 0

procedimento Destroi(ref L: ListaLinear)
    //nada a ser feito
```

Espaço:
θ(MAX_N)

Listas Lineares

- Todas as estruturas de dados possuirão dois procedimentos especiais:
  - `Controi`: chamado antes do primeiro uso com o propósito de inicializar as variáveis com os valores adequados
  - `Destroi`: chamado após o último uso com o propósito de desalocar recursos

Listas Lineares

- Revisitando o Exemplo 1:

```
var L: ListaLinear <Inteiro, Inteiro>
Constroi(L) //sempre presente implicitamente
Insere(L, 1, 1)
Insere(L, 2, 2)
Enumera(L) //saída: 1,2 ou 2,1
Remove(L, 1)
Enumera(L) //saída: 2
Destroi(L) //sempre presente implicitamente
```

Listas Lineares

- Tamanho (Ordenada ou Não):

```
função Tamanho(ref L: ListaLinear): Inteiro
   retornar (L.N)
```

Tempo:
θ(1)

Listas Lineares

- Enumera em Listas (Ordenada ou Não):

```
procedimento Enumera(ref L: ListaLinear)
   var i: Inteiro
   para i ← 1 até L.N faça
      escrever (L.Val[i].Chave)
```

Tempo:
θ(N)

Listas Lineares

- Busca em Lista Não-Ordenada:

```
função BuscaPosicao(ref L: ListaLinear, c: <TChave>):
                                                    Inteiro
    var i: Inteiro ← 1
    enquanto i ≤ L.N e L.Val[i].Chave ≠ c faça
        i ← i + 1
    se i ≤ L.N então
        retornar (i)
    senão
        retornar (0)
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Busca em Lista Não-Ordenada:

```
função BuscaEm(ref L: ListaLinear,
                  Pos: Inteiro): <TElem>
   retornar (L.Val[Pos].Elem)
```

Tempo:
θ(1)

Listas Lineares

- Busca em Lista Não-Ordenada:

```
função Busca(ref L: ListaLinear, c: <TChave>):
                                          <TElem>
   var Pos ← BuscaPosicao(L, c)
   se Pos = 0 então
      retornar "NULO"
   senão
      retornar (L.Val[Pos].Elem)
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Busca em Lista Ordenada (**Busca Binária**):

```
função BuscaPosicao(ref L: ListaLinear, c: <TChave>): Inteiro
    var inf, sup, m: Inteiro
    inf, sup ← 1, N
    enquanto inf ≤ sup faça
        m ← ⌊(inf/2 + sup/2)⌋
        se L.Val[m].Chave = c então
            retornar (m)
        senão se L.Val[m].Chave < c então
            inf ← m + 1
        senão
            sup ← m - 1
    retornar (0)
```

Listas Lineares

- Busca em Lista Ordenada (**Busca Binária**):

```
função BuscaPosicao(ref L: ListaLinear, c: <TChave>): Inteiro
    var inf, sup, m: Inteiro
    inf, sup ← 1, N
    enquanto inf ≤ sup faça
        m ← ⌊(inf/2 + sup/2)⌋
        se L.Val[m].Chave = c então
            retornar (m)
        senão se L.Val[m].Chave < c então
            inf ← m + 1
        senão
            sup ← m - 1
    retornar (0)
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(lg N)

Listas Lineares

- Busca em Lista Ordenada (**Busca Binária**):

```
função BuscaPosicao(ref L: ListaLinear, c: <TChave>,
                    ref PosIns: Inteiro): Inteiro
    var inf, sup, m: Inteiro
    inf, sup ← 1, N
    enquanto inf ≤ sup faça
        m ← ⌊(inf/2 + sup/2)⌋
        se L.Val[m].Chave = c então
            retornar (m)
        senão se L.Val[m].Chave < c então
            inf ← m + 1
        senão
            sup ← m - 1
    PosIns ← inf
    retornar (0)
```

Esta variante é útil para informar onde o elemento procurado deveria estar no caso da chave não ser encontrada

Listas Lineares

- Busca em Lista Ordenada:

```
função BuscaEm(ref L: ListaLinear,
                    Pos: Inteiro): <TElem>
   retornar (L.Val[Pos].Elem)
```

Tempo:
θ(1)

Listas Lineares

- Busca em Lista Ordenada:

```
função Busca(ref L: ListaLinear, c: <TChave>):
                                          <TElem>
   var Pos ← BuscaPosicao(L, c)
   se Pos = 0 então
      retornar "NULO"
   senão
      retornar (L.Val[Pos].Elem)
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(lg N)

Listas Lineares

- Inserção em Lista Não-Ordenada:

```
procedimento InsereEm(ref L: ListaLinear, Pos: Inteiro,
                              c: <TChave>, x: <TElem>)
    se L.N < MAX_N então
        L.Val[L.N+1], L.Val[Pos].Chave, L.Val[Pos].Elem,
            L.N ← L.Val[Pos], c, x, L.N+1
    senão
        Exceção("Overflow")
```

Tempo:
θ(1)

Listas Lineares

- Inserção em Lista Não-Ordenada:

```
procedimento Insere(ref L: ListaLinear, c: <TChave>,
                                    x: <TElem>)
    se L.N < MAX_N então
        L.Val[L.N+1].Chave, L.Val[L.N+1].Elem, L.N ←
            c, x, L.N+1
    senão
        Exceção("Overflow")
```

Tempo:
θ(1)

Listas Lineares

- Inserção em Lista Ordenada:

```
procedimento InsereEm(ref L: ListaLinear, Pos: Inteiro,
                              c: <TChave>, x: <TElem>)
    se L.N < MAX_N então
        para i ← L.N até Pos passo -1 faça
            L.Val[i+1] ← L.Val[i]
        L.N, L.Val[Pos].Chave, L.Val[Pos].Elem ←
            L.N+1, c, x
    senão
        Exceção("Overflow")
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Inserção em Lista Ordenada:

```
procedimento Insere(ref L: ListaLinear, c: <TChave>,
                                        x: <TElem>)
    var Pos, PosIns: Inteiro
    Pos ← BuscaPosicao(L, c, PosIns)
    se Pos = 0 então
        InsereEm(L, PosIns, c, x)
    senão
        Exceção("Chave existente")
```

Tempo:
Melhor Caso: θ(lg N)
Pior Caso: θ(N)

Listas Lineares

- Remoção em Lista Não-Ordenada:

```
função RemoveEm(ref L: ListaLinear,
                Pos: Inteiro): <TElem>
   se L.N > 0 então
      var x: <TElem> ← L.Val[Pos].Elem
      L.Val[Pos], L.N ← L.Val[N], L.N-1
      retornar (x)
   senão
      Exceção("Underflow")
```

Tempo:
θ(1)

Listas Lineares

- Remoção em Lista Não-Ordenada:

```
função Remove(ref L: ListaLinear, c: <TChave>):
                                          <TElem>
   var Pos: Inteiro ← BuscaPosicao(L, c)
   se Pos > 0 então
      retornar (RemoveEm(L, Pos))
   senão
      Exceção("Chave inexistente")
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Remoção em Lista Ordenada:

```
função RemoveEm(ref L: ListaLinear,
                    Pos: Inteiro): <TElem>
   var i: Inteiro
   se L.N > 0 então
      var x: <TElem> ← L.Val[Pos].Elem
      para i ← Pos até L.N-1 faça
         L.Val[i] ← L.Val[i+1]
      L.N ← L.N - 1
      retornar (x)
   senão
      Exceção("Underflow")
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Remoção em Lista Ordenada:

```
função Remove(ref L: ListaLinear, c: <TChave>):
                                          <TElem>
   var Pos: Inteiro ← BuscaPosicao(L, c)
   se Pos > 0 então
      retornar (RemoveEm(L, Pos))
   senão
      Exceção("Chave inexistente")
```

Listas Lineares

- Remoção em Lista Ordenada:

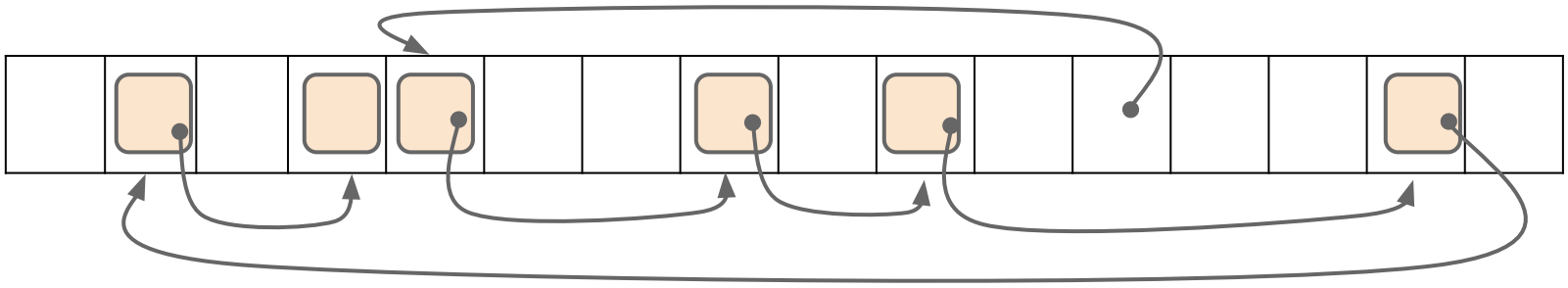
```
função Remove(ref L: ListaLinear, c: <TChave>):
                                          <TElem>
   var Pos: Inteiro ← BuscaPosicao(L, c)
   se Pos > 0 então
      retornar (RemoveEm(L, Pos))
   senão
      Exceção("Chave inexistente")
```

Tempo:
Melhor Caso: θ(lg N)
Pior Caso: θ(N)

Alocação Encadeada (elementos armazenados numa lista ligada)

Listas Lineares

- Uma ***lista encadeada*** (ou ***ligada***) é uma estrutura de dados vazia (ponteiro NULO) ou composta de um elemento especial chamado ***nó*** que contém um Elemento e um ponteiro para a lista ligada com o restante dos elementos.

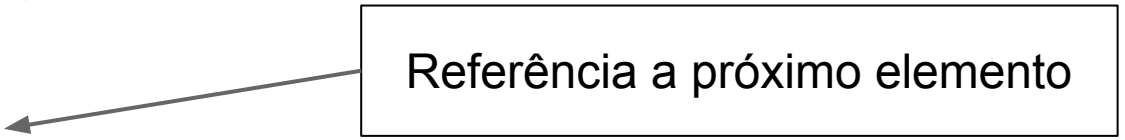


Listas Lineares

```
estrutura No <TChave, TElem>:
    Chave: <TChave>
    Elem: <TElem>
    Prox: ^No <TChave, TElem>

estrutura ListaLinear <TChave, TElem>:
    Inicio: ^No <TChave, TElem>
    N: Inteiro

procedimento Constroi(ref L: ListaLinear)
    L.Inicio, L.N ← NULO, 0
```



Listas Lineares

- Enumera em Listas (Ordenada ou Não):

```
procedimento Enumera(ref L: ListaLinear)
   var p: ^No
   p ← L.Inicio
   enquanto p ≠ NULO faça
      escrever (p^.Chave, p^.Elem)
      p ← p^.Prox
```

Tempo:
θ(N)

Listas Lineares

- Destruição (Ordenada ou Não):

```
procedimento Destroi(ref L: ListaLinear)
   var p, pprox: ^No
   pprox ← L.Inicio
   enquanto pprox ≠ NULO faça
      pprox, p ← pprox^.Prox, pprox
      desalocar (p)
```

Tempo:
θ(N)

Listas Lineares

- Busca em Lista Não-Ordenada:

```
função Busca(ref L: ListaLinear, c: <TChave>):
                                          <TElem>
   var p: ^No
   p ← L.Inicio
   enquanto p ≠ NULO E p^.Chave ≠ c faça
      p ← p^.Prox
   se p ≠ NULO então
      retornar (p^.Elem)
   senão
      retornar("NULO")
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Busca em Lista (Ordenada ou Não):

```
função BuscaEm(ref L: ListaLinear,
                    Pos: Inteiro): <TElem>
   p: ^No ← L.Inicio
   para i ← 2 até Pos faça
      p ← p^.Prox
   retornar (p^.Elem)
```

Tempo:
θ(Pos) = O(N)

Listas Lineares

- Busca em Lista Ordenada:

```
função Busca(ref L: ListaLinear, c: <TChave>):
                                              <TElem>
   var p: ^No ← L.Inicio
   enquanto p ≠ NULO E p^.Chave < c faça
      p ← p^.Prox
   se p ≠ NULO E p^.Chave = c então
      retornar (p^.Elem)
   senão
      retornar("NULO")
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Busca em Lista Ordenada:

```
função Busca(ref L: ListaLinear, c: <TChave>,
                                ref noAnt: ^No): <TElem>
   var p: ^No ← L.Inicio
   noAnt ← NULO
   enquanto p ≠ NULO E p^.Chave < c faça
      p, noAnt ← p^.Prox, p
   se p ≠ NULO E p^.Chave = c então
      retornar (p^.Elem)
   senão
      retornar("NULO")
```

Esta variante é útil para informar onde o elemento procurado deveria estar no caso da chave não ser encontrada

Listas Lineares

- Inserção em Lista Não-Ordenada:

```
procedimento Insere(ref L: ListaLinear,
                          c: <TChave>, x: <TElem>)
   var Novo: ^No
   alocar(Novo)
   Novo^.Chave, Novo^.Elem, Novo^.Prox,
      L.Inicio, L.N ←
      c, x, L.Inicio, Novo, L.N+1
```

Tempo:
θ(1)

Listas Lineares

- Inserção em Lista (Ordenada ou Não):

```
procedimento InsereEm(ref L: ListaLinear, Pos: Inteiro,
                                    c: <TChave>, x: <TElem>)
    var Novo: ^No, i: Inteiro ← 0, p: ^No ← L.Inicio
    alocar(Novo)
    Novo^.Chave, Novo^.Elem ← c, x
    se Pos = 1 então
        L.Inicio, Novo^.Prox ← Novo, L.Inicio
    senão
        p ← L.Inicio
        para i ← 1 até Pos - 2 faça
            p ← p^.Prox
        p^.Prox, Novo^.Prox ← Novo, p^.Prox
    L.N ← L.N+1
```

Tempo:
θ(Pos)

Listas Lineares

- Inserção em Lista Ordenada:

```
procedimento Insere(ref L: ListaLinear, c: <TChave>, x: <TElem>)
    var noAnt, Novo: ^No, v: <TElem>
    v ← Busca(L, c, noAnt)
    se v = "NULO" então
        alocar (Novo)
        Novo^.Chave, Novo^.Elem ← c, x
        se noAnt = NULO então
            Novo^.Prox, L.Inicio ← L.Inicio, Novo
        senão
            Novo^.Prox, noAnt^.Prox ← noAnt^.Prox, Novo
        L.N ← L.N+1
    senão
        Exceção("Chave existente")
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Remoção em Lista (Ordenada ou Não):

```
função Remove(ref L: ListaLinear, c: <TChave>): <TElem>
    var p, noAnt: ^No, x: <TElem>
    x ← Busca(L, c, noAnt)
    se x ≠ "NULO" então
        se noAnt = NULO então
            p, L.Inicio ← L.Inicio, L.Inicio^.Prox
        senão
            p, noAnt^.Prox ← noAnt^.Prox, noAnt^.Prox^.Prox
        desalocar (p)
        L.N ← L.N-1
        retornar (x)
    senão
        Exceção("Chave inexistente")
```

Tempo:
Melhor Caso: θ(1)
Pior Caso: θ(N)

Listas Lineares

- Remoção em Lista (Ordenada ou Não):

```
função RemoveEm(ref L: ListaLinear, Pos: Inteiro): <TElem>
    var p, noAnt: ^No, x: <TElem>
    se Pos = 1 então
        noAnt, x ← NULO, L.Inicio^.Elem
    senão
        noAnt ← L.Inicio
        para i ← 2 até Pos - 1 faça
            noAnt ← noAnt^.Prox
        x ← noAnt^.Prox^.Elem
    // segue agora como em Remove(), logo após a Busca()
```

Tempo:
θ(Pos)

Listas Lineares

- **Resumo**:

Tempo (Pior Caso)	Alocação Sequencial		Alocação Encadeada	
	Ordenada	Não-Ordenada	Ordenada	Não-Ordenada
Insere	θ(N)	θ(1)	θ(N)	θ(1)
Busca	θ(lg N)	θ(N)	θ(N)	θ(N)
Remove	θ(N)	θ(N)	θ(N)	θ(N)

Espaço	Alocação Sequencial	Alocação Encadeada
	θ(MAX_N)	θ(N)

Listas Lineares

- **Exercício**:

  Verifique se o uso de algum tipo particular de lista ((i) alocação sequencial/alocação encadeada, (ii) ordenada/não-ordenada) traz vantagem em relação à complexidade de tempo do algoritmo abaixo:

```
var N, k: Inteiro
ler (N)
var L: ListaLinear <Inteiro, Inteiro>
para i ← 1 até N faça
   ler (k)
   Insere(L, k, k*k)
ler (k)
escrever (Busca(k))
```

Listas Lineares

- **Exercício**:

  Verifique se o uso de algum tipo particular de lista ((i) alocação sequencial/alocação encadeada, (ii) ordenada/não-ordenada) traz vantagem em relação à complexidade de tempo do algoritmo abaixo:

```
var N, k: Inteiro
ler (N)
var L: ListaLinear <Inteiro, Inteiro>
para i ← 1 até N faça
   ler (k)
   Insere(L, k, k*k)
ler (k)
escrever (Busca(k))
```

seq./encad.+ não-ord.: $\theta(N)$
seq./encad.+ ord.: $O(N^2)$

Listas Lineares

- **Exercício**:

  Verifique se o uso de algum tipo particular de lista ((i) alocação sequencial/alocação encadeada, (ii) ordenada/não-ordenada) traz vantagem em relação à complexidade de tempo do algoritmo abaixo:

```
var k: Inteiro
var L: ListaLinear <Inteiro, Inteiro>
ler (k)
enquanto k > 0 faça
   Insere(L, k, k*k)
   ler (k)
para i ← 1 até Tamanho(L) faça
   ler (k)
   escrever (Busca(k))
```

Listas Lineares

- **Exercício**:

  Verifique se o uso de algum tipo particular de lista ((i) alocação sequencial/alocação encadeada, (ii) ordenada/não-ordenada) traz vantagem em relação à complexidade de tempo do algoritmo abaixo:

```
var k: Inteiro
var L: ListaLinear <Inteiro, Inteiro>
ler (k)
enquanto k > 0 faça
   Insere(L, k, k*k)
   ler (k)
para i ← 1 até Tamanho(L) faça
   ler (k)
   escrever (Busca(k))
```

encadeamento melhor (não há previsão de tamanho) + ord./não-ord.: $O(N^2)$

Listas Lineares

- **Exercício**:

  Verifique se o uso de algum tipo particular de lista ((i) alocação sequencial/alocação encadeada, (ii) ordenada/não-ordenada) traz vantagem em relação à complexidade de tempo do algoritmo abaixo:

```
var N, k: Inteiro
ler (N)
var L: ListaLinear <Inteiro, Inteiro>
para i ← 1 até N faça
   ler (k)
   Insere(L, k, k*k)
para i ← 1 até N^2 faça
   ler (k)
   escrever (Busca(k))
```

Listas Lineares

- **Exercício**:

  Verifique se o uso de algum tipo particular de lista ((i) alocação sequencial/alocação encadeada, (ii) ordenada/não-ordenada) traz vantagem em relação à complexidade de tempo do algoritmo abaixo:

```
var N, k: Inteiro
ler (N)
var L: ListaLinear <Inteiro, Inteiro>
para i ← 1 até N faça
   ler (k)
   Insere(L, k, k*k)
para i ← 1 até N² faça
   ler (k)
   escrever (Busca(k))
```

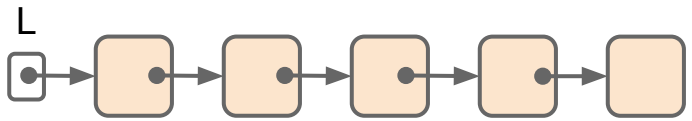
(i): sequencial (por causa do item (ii))
(ii): não-ordenada: $O(N^3)$
ordenada: $O(N^2 \lg N)$ (melhor)

Listas Lineares

- Outras variações de listas encadeadas:
  - **Sem Estrutura**:
    lista ao invés de ser uma estrutura consiste simplesmente de um ponteiro para o primeiro elemento (o número de elementos, neste caso, deve ser inferido por contagem dos elementos)

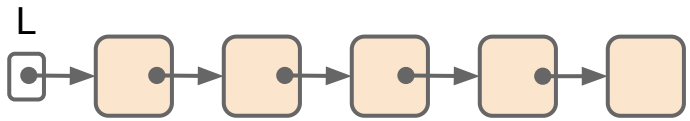
```
estrutura No <TChave, TElem>:
    Chave: <TChave>
    Elem: <TElem>
    Prox: ^No <TChave, TElem>

var L: ^No <Inteiro, Caracter>
//L é a lista linear encadeada
```

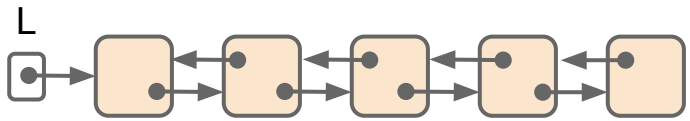


Listas Lineares

- Outras variações de listas encadeadas:
  - **Simplesmente Encadeada**:
    apenas ponteiro para próximo elemento

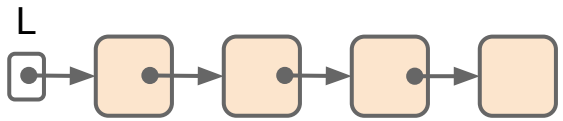


  - **Duplamente Encadeada**:
    ponteiros para elementos próximo e anterior

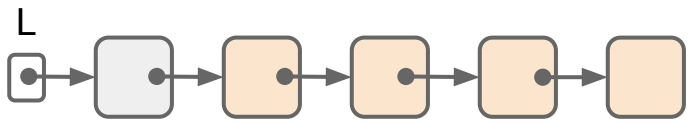


Listas Lineares

- Outras variações de listas encadeadas:
  - **Sem nó-cabeça:**
    L é um ponteiro para o primeiro elemento

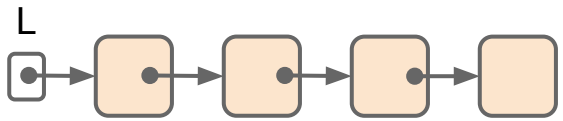


  - **Com nó-cabeça:**
    L é um ponteiro para um elemento que sempre existe (a "cabeça"), até quando a lista está vazia. Não deve ser considerado como elemento de fato

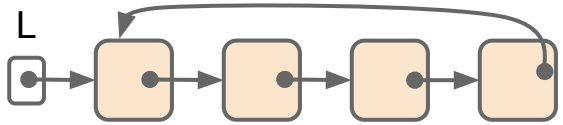


Listas Lineares

- Outras variações de listas encadeadas:
  - **não-Circular:**
    L termina quando, navegando pelos elementos, o ponteiro para o próximo elemento é vazio



  - **Circular:**
    L termina quando, navegando pelos elementos, o ponteiro para o próximo elemento é o mesmo de L



Exercícios

Exercícios

1. Reescrever os algoritmos de enumeração, busca, inserção, remoção em listas para considerar:
   a. Listas encadeadas clássica (consistindo de um ponteiro para `Nó`)
   b. Listas duplamente encadeadas
   c. Listas com nó cabeça
   d. Listas circulares

2. Dada uma matriz `B[1..N,1..N]: Inteiro` na qual cada linha e cada coluna está ordenada, isto é, se i < j então `B[k,i] ≤ B[k,j]` e `B[i,k] ≤ B[j,k]` para todo 1 ≤ k ≤ N, determinar se um elemento x dado como entrada pertence à matriz com as seguintes complexidades de tempo:
   a. $O(N^2)$
   b. O(N lg N)
   c. O(N)

Exercícios

3. Dada uma lista linear com alocação encadeada com inteiros como elementos (representando estes a própria chave), faça procedimentos ou funções que computem:
   a. o número de elementos positivos
   b. o maior elemento
   c. o último elemento
   d. o k-ésimo último elemento (k dado como entrada)
   e. remova todos os valores maiores que k (entrada)
   f. Verdadeiro ou Falso, indicando se os valores estão ordenados
   g. a média dos valores (suponha que a lista pode ser grande o suficiente para que a soma de todos os valores numa única variável inteira excede o maior inteiro possível de ser representado na linguagem)
   h. uma outra lista ligada com os valores da lista original sem repetições
   i. uma outra lista ligada, com o primeiro valor da lista original somado com o segundo, o segundo com o terceiro, etc.
   j. uma outra lista linear encadeada com todos os valores da lista original somados dois a dois
   k. uma outra lista linear encadeada com os elementos da lista original em ordem invertida

Exercícios

4. Dados dois naturais representados através de duas listas encadeadas L_1 e L_2, onde cada nó da lista contém um algarismo do número na ordem do menos significativo até o mais significativo, produzir uma lista encadeada S representando o número soma $L_1 + L_2$. Exemplo: **Entrada**: L_1: 2 → 3 → 7 → 9, L_2: 9 → 3 → 4; **Saída**: S: 1 → 7 → 1 → 0 → 1.

5. Dado um ponteiro p para um nó de uma lista, remover nó p em tempo:
   a. O(N), se a lista é simplesmente encadeada
   b. θ(1), se a lista é duplamente encadeada e com nó-cabeça

6. Dada uma lista simplesmente encadeada L circular, na qual o último nó aponta como próximo nó não necessariamente o primeiro, mas qualquer nó anterior, faça um algoritmo para detectar para qual nó o último nó aponta. Não há armazenado em L o número de elementos. Exemplo: **Entrada**: L: A → B → C → D → E → B; **Saída**: B. O algoritmo deve usar espaço auxiliar e tempo, respectivamente:
   a. O(N) e $O(N^2)$
   b. apenas dois ponteiros e θ(N)

Exercícios

7. Dado duas listas lineares ordenadas A e B com N elementos, determinar em tempo $\theta(N)$ o número de pares distintos de elementos x, y, onde x um elemento de A e y de B, tais que x e y possuam o mesmo valor

8. Dada uma lista linear com N elementos, criar um procedimento para remover elementos repetidos. Faça tal algoritmo com complexidades de espaço e tempo de, respectivamente,
   a. $O(N)$ e $O(N^2)$
   b. $\theta(1)$ e $O(N^2)$

9. Considere o problema de consultar um valor x inteiro em um vetor A com N inteiros distintos ordenado circularmente, ou seja, para algum $1 \leq k \leq N$, verifica-se que $A[k] < A[k+1] < A[k+2] < \cdots < A[N] < A[1] < A[2] < \cdots < A[k-1]$. Elabore um algoritmo que:
   a. retorne a posição do elemento x em A[1..N] ou retorne 0, caso x não esteja em A, em tempo $O(\lg N)$
   b. retorne o menor elemento em A[1..N] em tempo $O(\lg N)$
   c. no caso de A admitir valores duplicados, mostre que qualquer algoritmo de busca por elemento tem tempo $\Omega(N)$

Exercícios

10. Considere uma estrutura de dados ListaLinearAutoAjustavel, constituindo internamente de uma Lista Linear com alocação sequencial, com as seguintes características: (i) MAX_N = 1 inicialmente; (ii) a cada vez que a operação de inserir detecta *overflow*, ao invés de reportar erro, dobra-se o valor de MAX_N e recria-se a lista linear conservando-se os elementos já existentes, criando-se espaço assim para a nova inserção. Mostre que usando esta estratégia, o tempo amortizado para inserir N elementos é $\theta(N)$.

Exercícios

12. Dada uma sequência de conteúdos de células de uma planilha eletrônica, escrever o que deve ser impresso na posição de cada uma destas células. O conteúdo de cada célula pode ser um número, caso em que o valor a ser impresso referente a esta célula é o próprio número, ou o conteúdo pode ser uma fórmula, caso em que o valor a ser impresso representa o valor da fórmula calculada. Como entrada, serão dados o número M de linhas e o número N de colunas da planilha e o número R de células da planilha que possuem conteúdo não-vazio, seguido de R linhas de texto informando o valor de cada célula. O conteúdo de uma linha de texto é uma sequência de valores X Y T V X_1 Y_1 X_2 Y_2 … indicando que o valor sendo informado é referente a célula (X, Y) (X-ésima linha com a Y-ésima coluna), T representa o tipo do conteúdo ("V" = valor; "F" = fórmula), V é o valor numérico desta célula caso T="V", ou se T="F", a fórmula sendo informada é assumida ser (X, Y) = (X_1, Y_1) + (X_2, Y_2) + … + (X_V, Y_V). Como saída, o algoritmo deve imprimir R trios X Y V, que representa que na célula (X, Y) o valor correspondente é V. O algoritmo deve ter tempo O(R + V_S), onde V_S é a soma de todos os valores correspondentes ao parâmetro "V" em células com conteúdo fórmula.
Exemplo:

Entrada:	**Saída:**
4 5 8	2 1 7
2 1 F 1 3 1	3 1 7
3 1 V 7	1 2 28
1 2 F 3 2 1 2 3 4 4	2 3 14
2 3 F 2 3 4 4 4	1 4 21
1 4 F 2 2 3 2 5	2 5 7
2 5 F 1 3 4	3 4 7
3 4 F 1 4 4	4 4 7
4 4 F 1 2 1	

(Adaptado de http://uva.onlinejudge.org/external/1/196.pdf)