Listas

Algoritmos e Estruturas de Dados

2005/2006

FEUP

Tipo de Dados Abstracto

- TDA
  - conjunto de objectos + conjuntos de operações
  - constituem uma abstracção matemática
  - em C++ são implementados por classes; as operações são implementadas por membros-função *públicos*
  - Um TDA não especifica como as operações são implementadas, *apenas os seus efeitos*

    Exemplos: listas de objectos, filas, dicionários, ...

TAD : Listas

- Lista
  - sequência de objectos do mesmo tipo

    $A_0, A_1, A_2, \ldots, A_n$

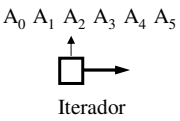
  - *lista vazia*: lista com zero elementos

  - Operações mais usuais:
    - criar uma lista vazia
    - adicionar/remover um elemento a uma lista
    - determinar a posição de um elemento na lista
    - determinar o comprimento (nº de elementos) de uma lista
    - concatenar duas listas

TAD: Iteradores

Para o tratamento de uma lista, é muitas vezes importante percorrer a lista, tratando os seus elementos um a um.

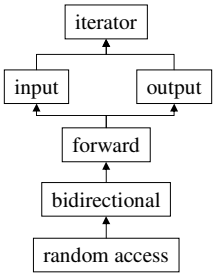
- Iterador
  - Abstracção que permite encapsular a informação sobre o estado desse processamento (i.e., a posição do elemento a processar)



  - Operações básicas:
    - iniciar
    - avançar
    - verificar se chegou ao fim

Iteradores (Standard Template Library - STL)

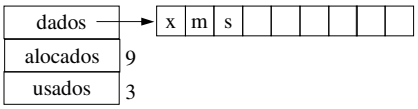
- "iterator"
  - construtor de cópia
  - operador afectação (=)
  - operadores prefixo e sufixo de incremento (++it, it++)
- "input": acesso a dados
  - operadores igualdade (==, !=)
  - desreferenciação (*it)
- "output": escrita de dados
  - desreferenciação e afectação (*it=el)
- "forward": percorrer num sentido,leitura/escrita dados
  - construtor sem argumentos
- "bidirectional": percorrer em ambos sentidos, leitura/escrita dados
  - operadores prefixo e sufixo de decremento (--it, it--)
- "random access": permite saltar de uma posição para outra
  - operadores de aritmética (+=, +, -=, -)
  - operadores de comparação (<, >, <=, >=)



(http://www.sgi.com/tech/stl/Iterators.html)

Listas: implementação baseada em vectores

- Implementação de listas
  - baseada em vectores
  - lista ligada
- Implementação de listas baseadas em vectores
  - Técnica: usar um vector para guardar os elementos da lista
  - Problema: comprimento da lista é variável; comprimento do vector é fixo
  - Solução:



Listas: implementação baseada em vectores

- Declaração da classe ***VList*** em C++ (secção privada)

```
template <class Object>
class VList {
private:
   Object * dados;
   int usados;
   int alocados;
   friend class VListItr<Object>;
// ...
};
```

Listas: implementação baseada em vectores

- Declaração da classe ***VList*** em C++ (secção pública)

```
template <class Object> class VList {
   // ....
public:
   VList (int size = 100);
   VList (const VList &origem);
   ~VList();
   bool isEmpty() const;
   void makeEmpty();
   VListItr<Object> first() const;
   VListItr<Object> beforeStart() const;
   void insert (const Object & x, const VListItr<Object> & p);
   void insert (const Object & x, int pos);
   VListItr<Object> find (const Object & x) const;
   void remove (const Object & rhs);
   const VList & operator = (const VList & rhs);
};
```

Listas: implementação baseada em vectores

- O iterador (secção privada)

```
template <class Object> class VListItr {
private:
   int posActual;            // índice ou -1 se antes do 1º elemento
   const VList<Object> & aLista;  // referência para a lista

   VListItr (const VList<Object> & v1, int pos = 0):
   aLista(v1), posActual(pos) {
      if ( pos > aLista.usados || pos < -1 ) throw BadIterator();
   } // construtor privado

   friend class VList<Object>;
// ....
};
```

Listas: implementação baseada em vectores

- O iterador (secção pública)

```
template <class Object> class VListItr {
public:
   bool isPastEnd() const {
      return ( aLista.usados == 0 || posActual >= aLista.usados );
   }

   void advance() { if ( !isPastEnd() ) posActual++; }

   const Object & retrieve() const {
      if ( isPastEnd() || posActual < 0 ) throw BadIterator();
      return aLista.dados[posActual];
   }
// ....
};
```

Listas: implementação baseada em vectores

- Utilização do iterador

```
template <class Object>
void printList (const VList<Object> & lst)
{
   cout << '[';
   for ( VListItr<Object> itr = lst.first(); !itr.isPastEnd(); itr.advance() )
      cout << itr.retrieve() << ' ';
   cout << ']';
}
```

Listas: implementação baseada em vectores

- classe VList : construtor e destrutor

```
template <class Object>
VList<Object>:: VList (int size = 100) {
   alocados = size;
   dados = new Object [size];
   usados = 0;
}

template <class Object>
VList<Object>:: ~VList () {
   delete [] dados;
}
```

Listas: implementação baseada em vectores

- classe VList : lista vazia

```
template <class Object>
bool VList<Object>:: isEmpty () const {
   return usados == 0;
}

template <class Object>
void VList<Object>:: makeEmpty () {
   delete [] dados;
   dados = new Object [alocados];
   usados = 0;
}
```

Listas: implementação baseada em vectores

- classe VList : iteradores para o início da lista
  - para o primeiro elemento

```
template <class Object>
VListItr<Object> VList<Object>:: first () const {
   return VListItr<Object> (*this, 0);      // não garante existência
}
```

  - para antes do primeiro elemento: posição fictícia, porque a inserção é realizada na posição à frente da indicada pelo iterador

```
template <class Object>
void VList<Object>:: beforeStart () const {
   return VListItr<Object> (*this, -1);      // antes do 1º elemento
}
```

Listas: implementação baseada em vectores

- classe VList : inserção
  - na posição à frente da indicada pelo iterador

```
template <class Object>
void VList<Object>:: insert (const Object & x, const VListItr<Object> &p) {
   if ( usados == alocados ) throw NoMoreSpace();
   for ( int i = usados-1; i > p.posActual; i-- )
            dados[i+1] = dados[i];
   dados[p.posActual+1] = x;
   usados++;
}
```

  - por índice

```
template <class Object> void VList<Object>:: insert (const Object & x, int pos = 0) {
   if ( pos > usados )  pos = usados;
   VListItr<Object> itr(*this, pos-1);
   insert(x, itr);
}
```

Listas: implementação baseada em vectores

- classe VList : pesquisa e remoção

```
template <class Object>
VListItr<Object> VList<Object>:: find (const Object & x) const {
   int i;
   for ( i = 0; i < usados && dados[i] != x; i++ ) { }
   return VListItr<Object> (*this, i);
}

template <class Object>
void VList<Object>:: remove (const Object & x) {
   VListItr<Object> p = find(x);
   if ( ! p.isPastEnd() ) {
      for ( int i = p.posActual; i < usados – 1; i++)
               dados[i] = dados[i+1];
      usados --;
   }
}
```

Listas: implementação baseada em vectores

- classe VList : operador de atribuição

```
template <class Object>
const VList<Object> &
VList<Object>:: operator = (const VList<Object> & rhs) {
   if ( alocados != rhs.alocados ) {
      delete [] dados;
      dados = new Object [rhs.alocados];
      alocados = rhs.alocados;
   }
   usados = rhs.usados;
   for ( int i = 0; i < usados; i++ )
      dados[i] = rhs.dados[i];
   return *this;
}
```

Listas: implementação baseada em vectores

- classe VList : construtor de cópia

```
template <class Object>
VList<Object>:: VList (const VList<Object> & rhs) {
   dados = new Object [rhs.alocados];
   usados = rhs.usados;
   alocados = rhs.alocados;
   for ( int i = 0; i < alocados; i++ )
      dados[i] = rhs.dados[i];
}
```

Listas: implementação baseada em vectores

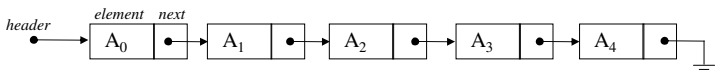
- Exemplo de utilização da classe ***VList***

```
VList<int> lista_inteiros;
lista_inteiros.insert(3, 0);
lista_inteiros.insert(9, 1);
lista_inteiros.insert(14, 2);
lista_inteiros.insert(11, 3);
lista_inteiros.insert(8, 5);
printList(lista_inteiros); cout << endl;
VListItr<int> itr = lista_inteiros.find(14);
lista_inteiros.insert(18,itr);
printList(lista_inteiros); cout << endl;
lista_inteiros.remove(3);
printList(lista_inteiros); cout << endl;
```

```
[ 3 9 14 11 8 ]
[ 3 9 14 18 11 8 ]
[ 9 14 18 11 8 ]
```

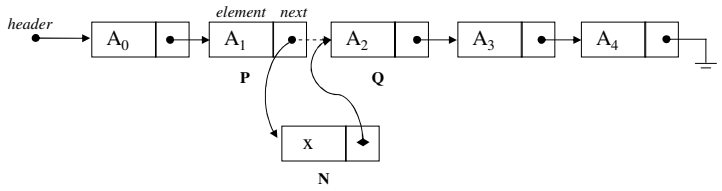
Listas ligadas

- Uma lista ligada é composta por nós. O nó possui dois campos:
  - o objecto a incluir na lista
  - um apontador para o elemento (nó) seguinte da lista



- Características:
  - espaço de memória ocupado pela lista evolui de acordo com as necessidades da aplicação
  - operações de inserção e remoção de elementos são realizadas por manipulação local dos apontadores: complexidade temporal *O(1)*

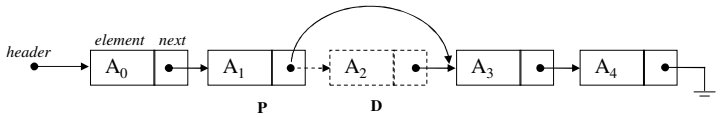
Inserir um elemento numa lista ligada




```
temp = P.next
P.next = N
N.next = temp
```

- O novo elemento (N) é colocado à frente de um dado elemento (P)
- O primeiro nó é um caso especial
- O último nó não é um caso especial.

Remover um elemento de uma lista ligada



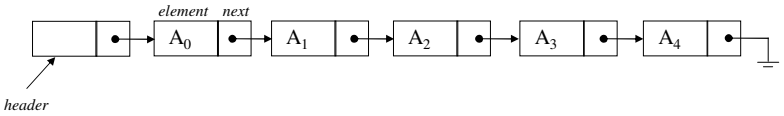

```
P.next = D.next
Destruir D
```

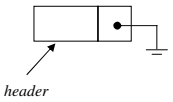
- Para remover o nó D é necessário aceder ao nó precedente P
- A remoção do primeiro nó é um caso especial
- A remoção do último nó não é um caso especial

Técnicas de implementação

Usar um *cabeçalho* (nó fictício) para simplificar a manipulação da lista

- o primeiro nó deixa de ser um caso especial





lista vazia

Listas: implementação listas ligadas

- A classe ***LListNode***

```
template <class Object>
class LListNode {
   LListNode (const Object & theElement = Object(), LListNode *n = 0)
         : element(theElement), next(n) { }

   Object element;
   LListNode *next;

   friend class LList<Object>;
   friend class LListItr<Object>;
};
```

Listas: implementação listas ligadas

- A classe ***LList*** (secção privada)

```
template <class Object>
class LList {
private:
   LListNode<Object> *header;    // nó fictício
   LListItr<Object> findPrevious(const Object & x) const;

public:
   // ...
};
```

Listas: implementação listas ligadas

- A classe ***LList*** (secção pública)

```
template <class Object> class LList {
public:
   LList();
   LList (const LList &rhs);
   ~LLIst();
   bool isEmpty() const;
   void makeEmpty();
   LListItr<Object> first() const;
   LListItr<Object> beforeStart() const;
   void insert(const Object & x, const LListItr<Object> & p);
   void insert(const Object & x, const int pos = 0);
   LListItr<Object> find(const Object & x) const;
   void remove(const Object & rhs);
   const LList & operator = (const LList & rhs);
private:    // ...
};
```

Listas: implementação listas ligadas

- A classe ***LListItr*** : iterador de listas ligadas (secção privada)

```
template <class Object>
class LListItr {
private:
   LListNode<Object> *current;

   LListItr(LListNode<Object> *theNode)
         : current(theNode) { };

   friend class LList<Object>;

public:
    // ...
};
```

Listas: implementação listas ligadas

- A classe ***LListItr*** : iterador de listas ligadas (secção pública)

```
template <class Object> class LListItr {
public:
   LListItr() : current(0) { };

   bool isPastEnd() const { return current == 0; }

   void advance() {
         if ( !isPastEnd() ) current = current->next;
   }

   const Object & retrieve() const {
         if ( isPastEnd() ) throw BadIterator();
         return current->element;
   }
private:
    // ...
};
```

Listas: implementação listas ligadas

- A classe ***LList*** : construtor e destrutor

```
template <class Object> LList<Object>::LList() {
   header = new LListNode<Object>;
}

template <class Object> LList<Object>::~LList() {
   makeEmpty();
   delete header;
}

template <class Object> void LList<Object>:: makeEmpty() {
   while ( !isEmpty() )
         remove(first().retrieve());
}
```

Listas: implementação listas ligadas

- A classe ***LList*** : estado e iteradores

```
template <class Object> bool LList<Object>::isEmpty() const {
   return header->next == 0;
}

template <class Object>
LListItr<Object> LList<Object>::beforeStart() const {
   return LListItr<Object>(header);
}

template <class Object>
LListItr<Object> LList<Object>::first() const {
   return LListItr<Object>(header->next);
}
```

Listas: implementação listas ligadas

- A classe ***LList*** : pesquisa

```
template <class Object>
LListItr<Object> LList<Object>::find(const Object & x) const {
   LListNode<Object> * itr = header->next;
   while ( itr != 0 && itr->element != x )
         itr = itr->next;
   return LListItr<Object>(itr);
}

template <class Object>
LListItr<Object> LList<Object>::findPrevious(const Object & x) const {
   LListNode<Object> * itr = header;
   while ( itr->next != 0 && itr->next->element != x )
         itr = itr->next;
   return LListItr<Object>(itr);
}
```

Listas: implementação listas ligadas

- A classe ***LList*** : remoção de elementos

```
template <class Object>
void LList<Object>::remove(const Object & x) {
   LListItr<Object> p = findPrevious(x);
   if ( p.current->next != 0 ) {
         LListNode<Object> * oldNode = p.current->next;
         p.current->next = p.current->next->next;
         delete oldNode;
   }
}
```

Listas: implementação listas ligadas

- A classe ***LList*** : inserção de elementos

```
template <class Object>
void LList<Object>::insert(const Object & x, const LListItr<Object> & p) {
   if ( p.current != 0 )
         p.current->next = new LListNode<Object>(x, p.current->next);
}

template <class Object>
void LList<Object>::insert(const Object & x, const int pos) {
   if ( pos < 0 ) throw BadPosition();
   LListNode<Object> * n = header;
   int posActual = 0;
   while ( n->next && posActual < pos) {
         posActual ++; n = n->next; }
   n->next = new LListNode<Object>(x, n->next);
}
```

Listas: implementação listas ligadas

- Exemplo de utilização da classe ***LList***

```
LList<int> lista_inteiros;
lista_inteiros.insert(3, 0); lista_inteiros.insert(9, 1);
lista_inteiros.insert(14, 2); lista_inteiros.insert(11, 3);
lista_inteiros.insert(8, 5);
printList(lista_inteiros); cout << endl;
LListItr<int> itr = lista_inteiros.find(14);
lista_inteiros.insert(18, itr);
printList(lista_inteiros); cout << endl;
lista_inteiros.remove(3);
printList(lista_inteiros); cout << endl;
LList<double> lista_reais;
LListItr<double> ditr = lista_reais.beforeStart();
lista_reais.insert(2.25, ditr); lista_reais.insert(9.95, ditr);
printList(lista_reais); cout << endl;
```

```
[ 3 9 14 11 8 ]
[ 3 9 14 18 11 8 ]
[ 9 14 18 11 8 ]
[ 9.95 2.25 ]
```

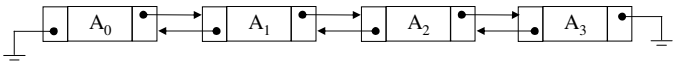
Listas (Standard Template Library - STL)

- class ***list***
- Alguns métodos:
  - iterator begin()
  - iterator end()
  - size_type size() const
  - bool empty() const
  - reference front()
  - reference back()
  - iterator insert(iterator p, cont T & e)
  - void push_front(const T & e)
  - void push_back(const T & e)
  - iterator erase(iterator p)
  - void pop_front()
  - void pop_back()
  - void clear()

(http://www.sgi.com/tech/stl/List.html)

Outros tipos de listas

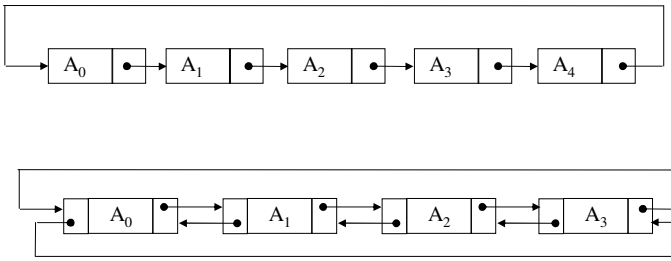
Listas duplamente ligadas



- Podem ser implementadas com dois nós "fictícios", um em cada extremo
- O iterador pode suportar eficientemente a operação *retreat*, com o efeito contrário a *advance*

Outros tipos de listas

Listas circulares



- Podem ser implementadas com um nó "fictício", que indica o primeiro elemento da lista

Aplicações de listas

- Polinómios esparsos
  - Polinómios de grau elevado, mas com poucos termos, p.ex:

    $3x^{1000} + 4x^{200} + 1$
  - A representação baseada em vectores desperdiça espaço de memória
    - Porque requer espaço de memória proporcional ao grau do polinómio e não ao número de termos
  - Uma implementação baseada em listas ligadas aproveita melhor o espaço de memória
    - O termo de maior grau é o primeiro da lista
    - A lista é mantida ordenada por grau (ordem decrescente)
    - Classes *Polinomio* e *Termo*

Aplicações de Listas

- A classe ***Termo***

```
class Termo
{
public:
   Termo (double coef=0.0, int pot=0) : coeficiente(coef), potencia(pot)
   { };
   double avaliar(double x);
   double coeficiente;
   int potencia;
};
Termo operator *(const Termo &t1, const Termo &t2);
Termo operator +(const Termo &t1, const Termo &t2);
bool operator ==(const Termo &t1, const Termo &t2);
bool operator !=(const Termo &t1, const Termo &t2);
ostream & operator <<(ostream &out, const Termo &val);
```

Aplicações de Listas

- A classe ***Termo*** (operações com termos)

```
double Termo::avaliar(double x)
{ return coeficiente*pow(x, (double) potencia);  }

Termo operator *(const Termo &t1, const Termo &t2)
{
   Termo resultado(t1.coeficiente * t2.coeficiente,   t1.potencia + t2.potencia);
   return resultado;
}

Termo operator +(const Termo &t1, const Termo &t2)
{
   if (t1.potencia != t2.potencia) throw ErroInterno();
   Termo resultado(t1.coeficiente + t2.coeficiente, t1.potencia);
   return resultado;
}
```

Aplicações de Listas

- A classe ***Polinomio*** (secção pública)

```
class Polinomio
{
public:
   Polinomio() { };
   Polinomio(const Polinomio &p);
   Polinomio(Termo &t);
   Polinomio(double coef_um, double coef_zero);
   Polinomio(double coef_dois, double coef_um, double coef_zero);

   void operator +=(const Polinomio &p);
    void operator +=(const Termo &t);
    void operator *=(const Termo &t);
   double avaliar(double x);
   // ...
};
```

Aplicações de Listas

- A classe ***Polinomio*** (secção privada)

```
class Polinomio
{
private:
   LList<Termo> termos;

   friend Polinomio operator *(const Polinomio &p, const Polinomio &q);
   friend Polinomio operator +(const Polinomio &p, const Polinomio &q);
   friend ostream & operator <<(ostream &out, const Polinomio &p);

   // ...
};
```

Aplicações de Listas

- A classe ***Polinomio*** (operações com polinómios)

```
double Polinomio:: avaliar(double x) {
   double sum = 0.0;
   LListItr<Termo> itr = termos.first();
   for ( ; !itr.isPastEnd(); itr.advance() )
       sum += itr.retrieve().avaliar(x);
   return sum;
}

void Polinomio::operator *=(const Termo &t) {
   LListItr<Termo> itr = termos.first();
   for ( ; !itr.isPastEnd(); itr.advance() )
       itr.retrieve() = itr.retrieve() * t;
}
```

Aplicações de Listas

- A classe ***Polinomio*** (operações com polinómios)

Uso de dois iteradores consecutivos

```
void Polinomio::operator +=(const Termo &t) {
   LListItr<Termo> itr = termos.beforeStart(), nitr = termos.first();
   while ( !itr.isPastEnd() ) {
       if ( nitr.isPastEnd() || nitr.retrieve().potencia < t.potencia ) {
           termos.insert(t, itr);
           break;
        } else if ( nitr.retrieve().potencia == t.potencia ) {
           itr.retrieve().coeficiente += t.coeficiente;
           break;
        } else {
           itr.advance();
           nitr.advance();
        }
    }
}
```

Aplicações de Listas

- A classe ***Polinomio*** (operações com polinómios)

```
Polinomio operator *(const Polinomio &p, const Termo &t) {
   Polinomio result(p);
   result *= t;
   return result;
}

Polinomio operator *(const Polinomio &p, const Polinomio &q) {
   Polinomio result;
   LListItr<Termo> itr = p.termos.first();
   for ( ; !itr.isPastEnd(); itr.advance() )
       result += q * itr.retrieve();
   return result;
}
```

Aplicações de Listas

- Teste da classe ***Polinomio***

```
Termo t(11.5, 2);
Polinomio p1(3.5, 2, 1), p2(2, 8), p3, p4;
cout << t << endl;
cout << "p1: " << p1 << endl << "p2: " << p2 << endl;
cout << "p2(1): " << p2.avaliar(1) << endl;
Polinomio temp = p1 + p2;
cout << "p1+p2: " << temp << endl;
cout << "(p1+p2) (1): " << temp.avaliar(1) << endl;
cout << "p1*p2: " << p1*p2 << endl;
p3 += Termo(10,3); p3 += Termo(2,100); p3 += Termo(1,50);
cout << "p3: " << p3 << endl;
cout << p3(1): " << p3.avaliar(1) << endl;
cout << "p3^2: " << p3*p3 << endl;
p4 += Termo(1,1000); p4 += Termo(1,0);
cout << "p4: " << p4 << endl;
cout << "p4^3: " << p4*p4*p4 << endl;
```

Aplicações de Listas

- Teste da classe ***Polinomio***

resultados:

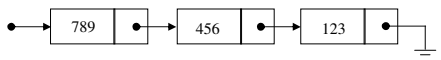
```
11.5x^2
p1: 3.5x^2 + 2x + 1
p2: 2x + 8
p1(1): 10
p1+p2: 3.5x^2 + 4x + 9
(p1+q1)(1): 16.5
p1*p2: 7x^3 + 32x^2 + 18x + 8
p3: 2x^100 + 1x^50 + 10x^3
p3(1): 13
p3^2: 4x^200 + 4x^150 + 40x^103 + 1x^100 + 20x^53 + 100x^6
p4: 1x^1000 + 1
p4^3: 1x^3000 + 3x^2000 + 3x^1000 +1
```

Aplicações de listas

- Números naturais "ilimitados"
  - O número 123456789 pode ser representado em base 1000 como

$$123 \times 1000^2 + 456 \times 1000^1 + 789 \times 1000^0$$

    - Cada um dos coeficientes está entre 0 e 999
  - Representar cada número como uma sequência de coeficientes ("digitos" em base 1000). Implementação baseada em listas ligadas:
    - O grupo menos significativo é o primeiro da lista
    - Classe *NumNatural*



Aplicações de Listas

- A classe ***NumNatural***

```
class NumNatural {
public:
   NumNatural(int n = 0) ;
   NumNatural(const NumNatural &n): digitos(n.digitos) { }
   NumNatural & operator =(const NumNatural &n) { digitos = n.digitos; }
   void output(ostream &out) const;
   void operator +=(const NumNatural &n);
private:
   LList<int> digitos;
   void rec_output(ostream &out, LListItr<int> &itr) const;
   static const int modulo;
};
NumNatural operator+ (const NumNatural &n, const NumNatural &m);
ostream & operator <<(ostream &out, const NumNatural &n);
```

Aplicações de Listas

- Como somar ?
  - Para cada grupo de dígitos (do menos significativo para o mais significativo):
    - somar os grupos (coeficientes) correspondentes e o transporte do grupo anterior
    - o novo valor é igual a *soma%1000*
    - o transporte para o grupo é igual a *soma/1000* (divisão inteira)
  - Cuidados a ter:
    - os números a somar tÊm geralmente comprimento diferente
    - a soma pode ter comprimento superior às duas parcelas

Aplicações de Listas

- A classe ***NumNatural*** (soma de números naturais)

```
void NumNatural::operator += (const NumNatural &n)  {
   int soma, nd, transporte = 0;
   LListItr<int> itr = digitos.first();
   LListItr<int> nitr = n.digitos.first();
   while ( true ) {
      if ( nitr.isPastEnd() && transporte == 0 ) break;
      if ( nitr.isPastEnd() ) nd = 0;
      else nd = nitr.retrieve();
      soma = itr.retrieve() + nd + transporte;
      transporte = soma/modulo;
      itr.retrieve() = soma % modulo;
      nitr.advance();
      if ( itr.isLast() ) break;
      else itr.advance();
   }
   // ... continua
```

Aplicações de Listas

- A classe ***NumNatural*** (soma de números naturais)

```
// continuação
   if ( transporte > 0 || !nitr.isPastEnd() ) {   // o comprimento do nº pode aumentar
      if ( nitr.isPastEnd() )
         nd = 0;
      else
         nd = nitr.retrieve();
      soma = transporte + nd;
      transporte = soma/modulo;
      digitos.insert(soma%modulo, itr);
      itr.advance();
      nitr.advance();
   }
}
```