

-
-
-
-
-
-
-

Listas

Algoritmos e Estruturas de Dados

2019/2020



-
-
-
-
-
-
-
-

Tipo de Dados Abstrato

- TDA
 - conjunto de objetos + conjunto de operações
 - constituem uma abstração matemática (dados são genéricos e não específicos)
 - **não especifica como as operações** são implementadas, *apenas os seus efeitos*
 - Implementação em C++:
 - classes genéricas
 - operações: membros-função públicos

Exs de TDAs: listas de objetos, filas, dicionários, ...

Exs de operações: comparar objetos, procurar um objeto, ...

TAD : Listas

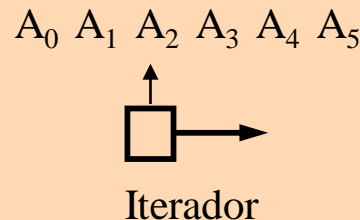
- Lista
 - sequência de objetos do mesmo tipo
$$A_0, A_1, A_2, \dots, A_n$$
 - *lista vazia*: lista com zero elementos
 - Operações mais usuais:
 - criar uma lista vazia
 - adicionar/remover um elemento a uma lista
 - determinar a posição de um elemento na lista
 - determinar o comprimento (nº de elementos) 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

- objeto apontador para um elemento de certos TDAs
- abstração que permite encapsular a informação sobre o estado do processamento do TDA (i.e., a posição do elemento a processar)



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

TAD: Iteradores

Iterador

- associa-se a um Tipo de Dados Abstractos ou a uma sua implementação
- *Exemplo* de uso de iteradores em vetores:
 - considere o vetor **nomes** (vetor de strings: nomes de pessoas, p.ex)
 - procurar o nome "Luis Silva" no vetor **nomes**

associa a um TDA

```
vector<string> nomes;  
// ... adicao de varios nomes no vetor  
vector<string>::iterator it;  
for (it=nomes.begin(); it != nomes.end(); it++)  
    if (*it == "Luis Silva")  
        cout << "Luis Silva encontrado!";
```

coloca-se em início

elemento "iterado"

se ultrapassa fim

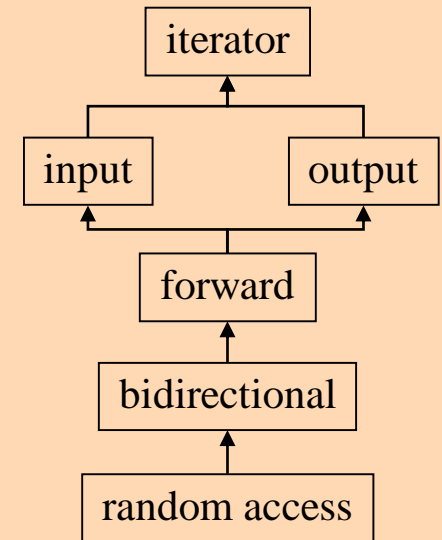
passa a iterar elemento seguinte

- mais informação sobre os iteradores C++ STL:

- <http://www.sgi.com/tech/stl/Iterators.html>
- <http://www.cppreference.com/iterators.html>

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 (<, >, <=, >=)



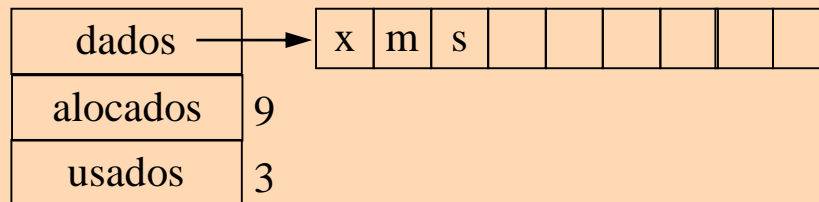
(<http://www.sgi.com/tech/stl/Iterators.html>)

Listas: implementação

- Técnicas artesanais de implementação de listas
 - baseada em *arrays*
 - baseada em apontadores:
 - listas ligadas
 - listas circulares
 - listas duplamente ligadas

Listas: implementação baseada em arrays

- Implementação de listas baseadas em *arrays*
 - os elementos são guardados num *array*
 - o tamanho do *array* (nº de elementos) exige monitorização constante
 - pesquisa, inserção e remoção de elementos:
 - operações de complexidade temporal $O(tamanho)$
 - Solução:



Listas: implementação baseada em arrays

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

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

iterador

Listas: implementação baseada em arrays

- 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 arrays

- O iterador (secção privada)

iterador

```
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>;  
    // ....  
};
```



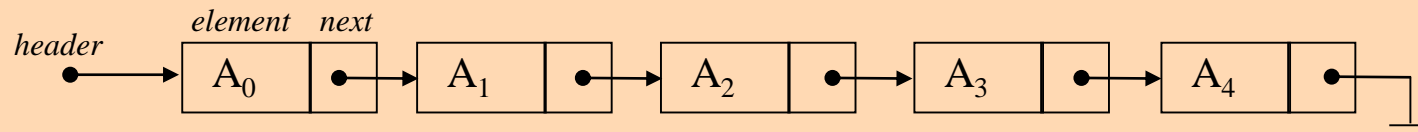
Listas: implementação baseada em arrays

- 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 listas ligadas

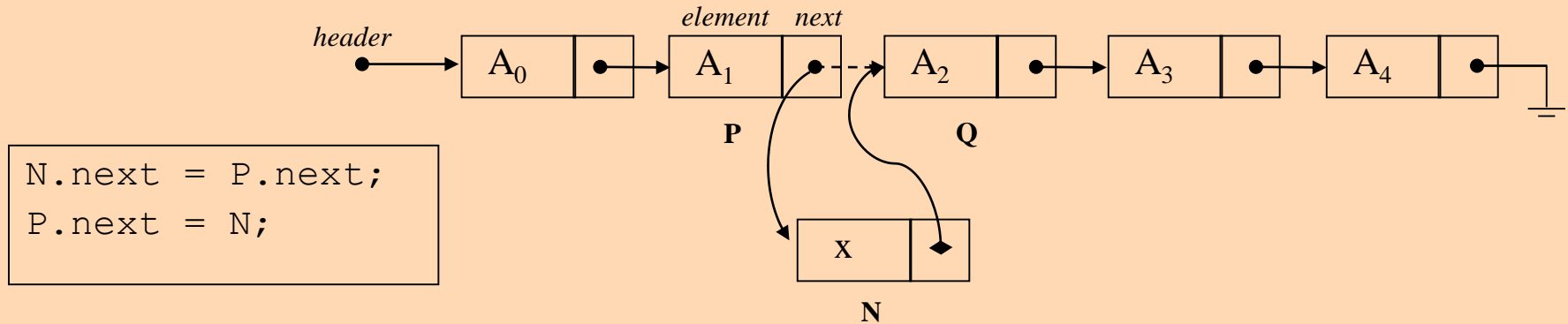
- Uma lista ligada é composta por **nós**. O nó possui dois campos:
 - o objeto a incluir na lista
 - um apontador para o elemento (nó) seguinte da lista



- tamanho da lista varia facilmente por alocação dinâmica
- pode ter ou não um nó especial (cabeçalho)
- inserção e remoção de elementos:
 - operações de complexidade temporal $O(1)$
- pesquisa de elementos:
 - operação de complexidade temporal $O(tamanho)$

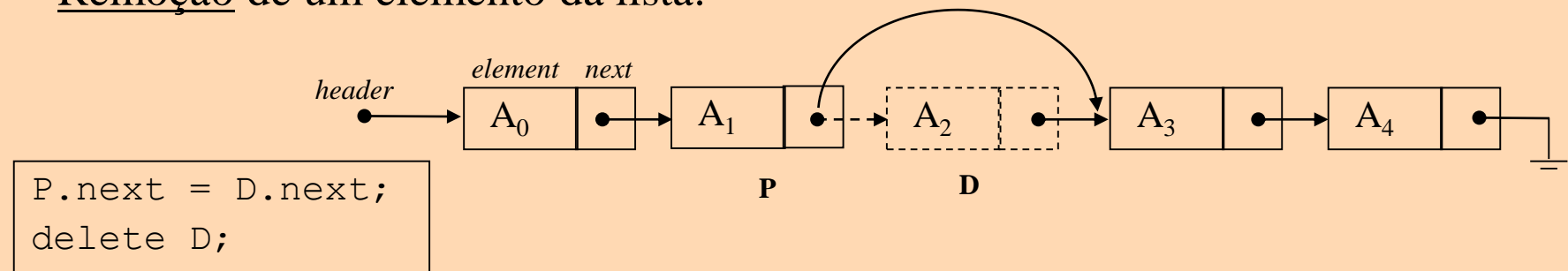
Listas: implementação baseada em listas ligadas

- Inserção de um elemento na lista:



- O primeiro nó é um caso especial

- Remoção de um elemento da lista:

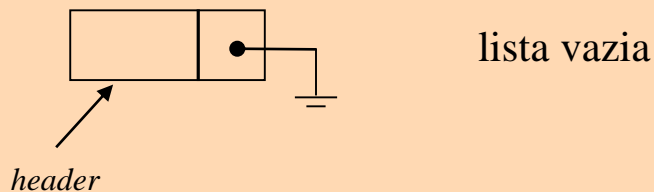
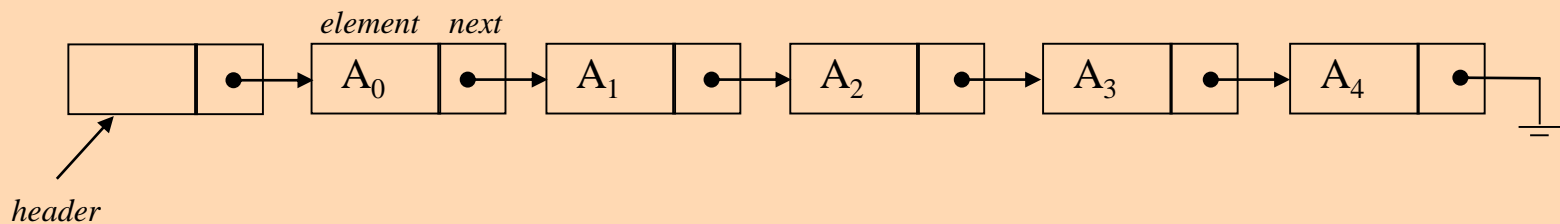


- O primeiro nó é um caso especial

Técnicas de implementação

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

- o primeiro nó deixa de ser um caso especial



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);  
    ...  
};
```

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>;
// ...
};
```



Listas: implementação listas ligadas

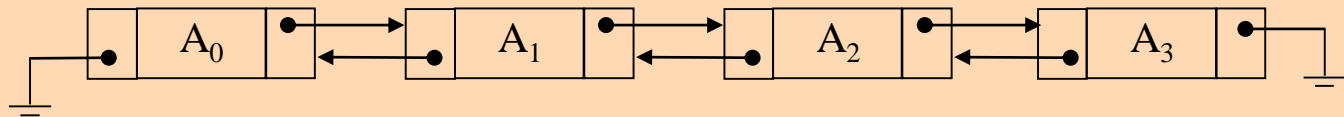
- A classe ***LListItr*** : iterador de listas ligadas (secção pública)

```
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;  
    }  
    // ...  
};
```

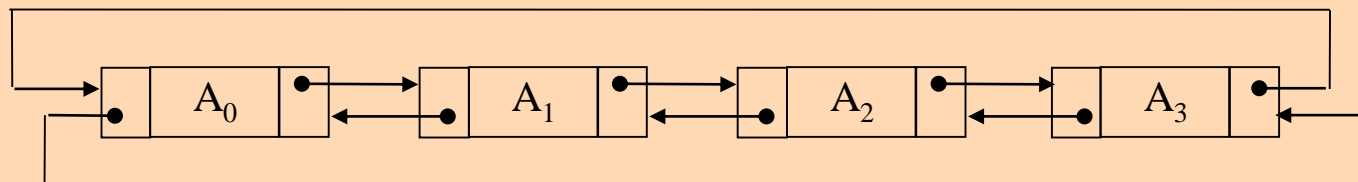
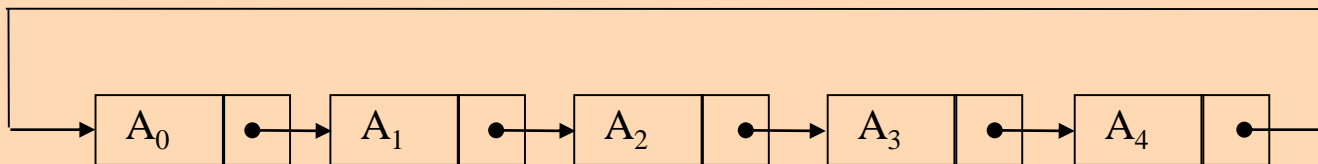


Outros tipos de listas

- Lista duplamente ligada
 - pode ter ou não nós especiais (cabeçalho e rodapé)



- Lista circular (simplesmente ou duplamente ligada)
 - pode ou não ter nós especiais (cabeçalho e rodapé)



Classe `list` (STL)

- classe `list` em STL:
 - *sequence* que pode ser percorrida nos dois sentidos: “para a frente” ou “para trás”
 - *sequence: container* de tamanho variável com elementos dispostos linearmente
 - *container*: objeto que armazena outros objetos (elementos)
 - suporta métodos de acesso aos elementos
 - tem iterador associado
 - implementada como lista duplamente ligada

consultar:

- <http://www.sgi.com/tech/stl/List.html>
- <http://www.cppreference.com/cpp/list>

Classe `list` (STL)

Alguns métodos de *list* (STL):

- iterator **begin()**
- iterator **end()**
- `size_type` **size()** `const`
- `bool` **empty()** `const`
- reference **back()**
- reference **front()**
- iterator **insert**(iterator *p*, `const T & e`) // insere *e* antes de *p*, retorna iterator para *e*
- `void` **push_back**(`const T & e`)
- `void` **push_front**(`const T & e`)
- iterator **erase**(iterator *p*) // retorna iterator para o elemento seguinte ao removido
- `void` **pop_front()**
- `void` **pop_back()**
- `void` **clear()**
- `void` **sort()**

Classe `list` (STL)

e o algoritmo **`sort`** ()

`void sort(iterator start, iterator end);`

`void sort(iterator start, iterator end, StrictWeakOrdering cmp);`

- class `list` **não** pode usar o algoritmo **`sort`** ()
- o algoritmo **`sort`** () da STL funciona com *Random Access Iterators* e não com *Bidirectional Iterators*
- a classe `list` usa *Bidirectional Iterators*
- mas `list` tem função-membro **`sort`** ()

e o algoritmo **`find`** ()

`iterator find(iterator start, iterator end, const TYPE& val);`

`iterator find_if(iterator start, iterator end, Predicate up);`

- class `list` pode usar o algoritmo **`find`** ()

Aplicação

- Polinómios esparsos

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

$$3x^{1000} + 4x^{200} + 1$$

- Polinómio de grau n : $P_n(x) = a_{n-1}x^{n-1} + \dots + a_3x^3 + a_2x^2 + a_1x + a_0$

- representar polinómio: lista de termos
- termo i : $a_i x^i$ ($= \text{coeficiente} * \text{base}^{\text{expoente}}$)
- operar com polinómios

- $P1 + P2$

- $k P1$

- $P1 * P2$

- avaliar polinómio: $P(x)$

Aplicação

– Representação baseada em arrays:

- *array* contém os coeficientes
- posição é o grau do termo
- desperdiça espaço de memória

– espaço é proporcional ao grau e não ao n° de termos!

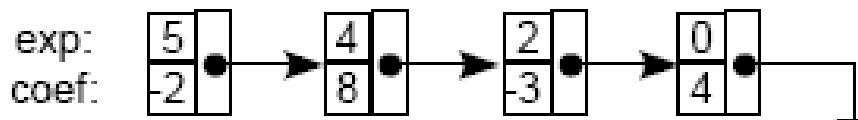
$-2x^5 + 8x^4 - 3x^2 + 4$ como *array*:

	4		-3		8	-2
grau:	0	1	2	3	4	5

– Representação baseada em listas ligadas:

- aproveita melhor o espaço de memória
- lista de termos [pares (coeficiente, expoente)]
- termo de maior grau é o primeiro da lista
- a lista é mantida ordenada por grau (ordem decrescente)

$-2x^5 + 8x^4 - 3x^2 + 4$ como *lista*



Aplicação

- A classe **Termo**

```
class Termo
{
public:
    double coeficiente;
    int potencia;

    Termo (double coef=0.0, int pot=0): coeficiente(coef),
                                         potencia(pot) { };

    double avaliar(double x);
};

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



Aplicação

- Operações com termos

```
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);
```

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

```
Termo operator +(const Termo &t1, const Termo &t2) {  
    if (t1.potencia != t2.potencia) {  
        cout << "operacao de soma impossivel";  
        Termo res(t1.coeficiente,t1.potencia); return res; }  
    Termo resultado(t1.coeficiente + t2.coeficiente, t1.potencia);  
    return resultado;
```

Aplicação

- A classe `Polinomio`

```
class Polinomio {  
public:  
    list<Termo> termos;  
  
    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);  
};
```

Aplicação

```
Polinomio::Polinomio(const Polinomio &p) {
    list<Termo>::const_iterator itr = p.termos.begin();
    list<Termo>::const_iterator itre = p.termos.end();
    while ( itr != itre ) {
        double coef = itr->coeficiente;
        int pot = itr->potencia;
        termos.push_back(Termo(coef,pot));
        itr++;
    }
}

Polinomio::Polinomio(double coef_um, double coef_zero) {
    Termo t1(coef_um,1); Termo t0(coef_zero,0);
    termos.push_front(t0);
    termos.push_front(t1);
}
```

Aplicação

```
Polinomio::Polinomio(double coef_dois, double coef_um,
                    double coef_zero) {

    Termo t2(coef_dois,2);
    Termo t1(coef_um,1); Termo t0(coef_zero,0);
    termos.push_front(t0); termos.push_front(t1);
    termos.push_front(t2);
}

double Polinomio:: avaliar(double x) {
    double sum = 0.0;
    list<Termo>::iterator itr = termos.begin();
    list<Termo>::iterator itre = termos.end();
    for ( ; itr!=itre ; itr++ )
        sum += itr->avaliar(x);
    return sum;
}
```



Aplicação

```
void Polinomio::operator +=(const Termo &t) {
    list<Termo>::iterator itr = termos.begin();
    list<Termo>::iterator itre = termos.end();
    while ( itr != itre ) {
        if ( itr->potencia < t.potencia ) {
            termos.insert(itr,t);
            return;
        }
        else if (itr->potencia == t.potencia ) {
            itr->coeficiente += t.coeficiente;
            return;
        }
        else itr++;
    }
    termos.insert(itr,t);
}
```


Aplicação

```
void Polinomio::operator +=(const Polinomio &p) {  
    list<Termo>::const_iterator itr = p.termos.begin();  
    list<Termo>::const_iterator itre = p.termos.end();  
    while (itr != itre) {  
        (*this) += (*itr);  
        itr++;  
    }  
}
```

```
void Polinomio::operator *=(const Termo &t) {  
    list<Termo>::iterator itr = termos.begin();  
    list<Termo>::iterator itre = termos.end();  
    for ( ; itr != itre ; itr++ )  
        (*itr) = (*itr) * t;  
}
```

Aplicação

```
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;  
    list<Termo>::const_iterator itr = p.termos.begin();  
    list<Termo>::const_iterator itre = p.termos.end();  
    for ( ; itr != itre; itr++ )  
        result += q * (*itr);  
    return result;  
}
```

Aplicação

```
ostream & operator <<(ostream &out, const Polinomio &p)
{
    list<Termo>::const_iterator itr = p.termos.begin();
    list<Termo>::const_iterator itre = p.termos.end();
    while (itr != itre) {
        out << (*itr) << " + " ;
        itr++;
    }
    out << "\n";
    return out;
}
```

Aplicação

```
int main() {
    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.avalciar(1) << endl;
    Polinomio temp = p1 + p2;
    cout << "p1+p2: " << temp << endl;
    cout << "(p1+p2) (1): " << temp.avalciar(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.avalciar(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;
}
```



Aplicação

- Resultado:

$$11.5x^2$$

$$p1: 3.5x^2 + 2x + 1$$

$$p2: 2x + 8$$

$$p2(1): 10$$

$$p1+p2: 3.5x^2 + 4x + 9$$

$$(p1+p2)(1): 16.5$$

$$p1*p2: 7x^3 + 32x^2 + 18x + 8$$

$$p3: 2x^{100} + 1x^{50} + 10x^3$$

$$p3(1): 13$$

$$p3^2: 4x^{200} + 4x^{150} + 40x^{103} + 1x^{100} + 20x^{53} + 100x^6$$

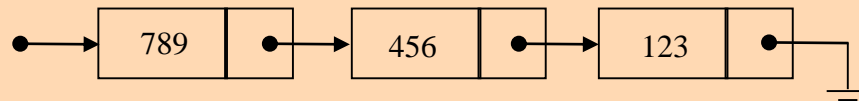
$$p4: 1x^{1000} + 1$$

$$p4^3: 1x^{3000} + 3x^{2000} + 3x^{1000} + 1$$



Aplicação 2

- 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 (“dígitos” em base 1000). Implementação baseada em listas:
 - O grupo menos significativo é o primeiro da lista
 - Classe *NumNatural*



Aplicação 2

- A classe `NumNatural`

```
class NumNatural {  
    list<int> digitos;  
    void rec_output(ostream &out,  
                   list<int>::iterator &itr) const;  
    static const int modulo;  
  
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);  
};
```



Aplicação 2

```
NumNatural operator+ (const NumNatural &n,  
                      const NumNatural &m);  
ostream & operator <<(ostream &out, const NumNatural &n);
```

- 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

Aplicação 2

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

```
void NumNatural::operator += (const NumNatural &n) {
    int soma, nd, transporte = 0;
    list<int>::iterator itr = digitos.begin();
    list<int>::iterator itre = digitos.end();
    list<int>::iterator nitr = n.digitos.begin();
    list<int>::iterator nitre = n.digitos.end();
    while ( true ) {
        if ( itr == itre ) break;
        if ( nitr == nitre && transporte == 0 ) break;
        if ( nitr == nitre ) nd = 0; else nd = *nitr;
        soma = *itr + nd + transporte;
        transporte = soma/modulo;
        (*itr) = soma % modulo;
        nitr++; itr++;
    }
    // ... continua
```

Aplicação 2

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

```
// continuação
while ( transporte > 0 || nitr != nitre ) {
    // o comprimento do n° pode aumentar
    if ( nitr == nitre )
        nd = 0;
    else
        nd = *nitr;
    soma = transporte + nd;
    transporte = soma/modulo;
    digitos.insert(itr,soma%modulo);
    itr++;
    nitr++;
}
}
```