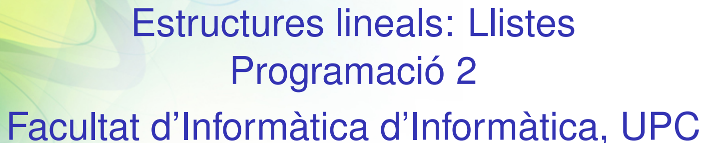


Estructures lineals: Llistes
Programació 2
Facultat d’Informàtica d’Informàtica, UPC

Professorat de PRO2

Primavera 2024

- Aquestes transparències **no** substitueixen els apunts de l’assignatura, els complementen

1. Llistes
2. Sets
3. Maps

Llistes

Les llistes ens ofereixen operacions per a fer:

- Recorreguts seqüencials de tots els elements
- Inserció d'un element nou a qualsevol punt de la seqüència
- Eliminació d'un element qualsevol
- Concatenació

TOTES LES OPERACIONS DE LLISTES SON CONSTANTS.

Iteradors

- El mecanisme que permet fer això amb les `list` de la STL són els iteradors
- Un *iterador* és un objecte que designa (marca, apunta, referencia) un element d'una llista o un altre contenidor
- Operacions sobre iteradors:
  - Avançar al següent element: `++it`
  - Retrocedir a l'anterior: `--it`
  - Comparar iteradors: `it1==it2`, `it1!=it2`
  - Accedir a l'objecte designat: `*it`

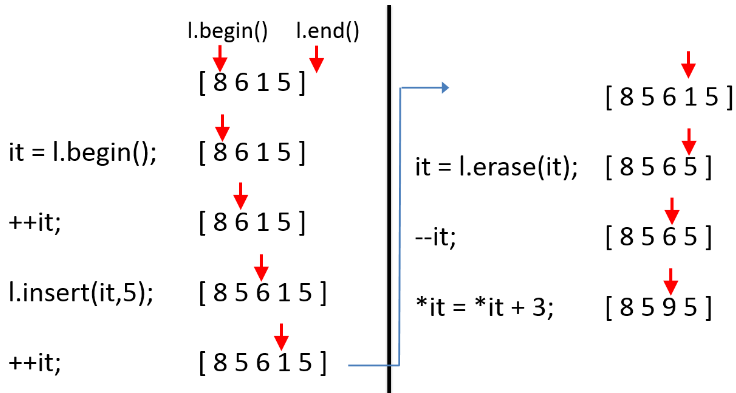
Iteradors

- Llistes i iteradors “treballen” coordinadament
- Operacions de llistes amb iteradors:
  - $L.\mathtt{insert}(it, x)$: insereix a la llista L un nou element x com a predecessor de l’element apuntat per it
  - $it = L.\mathtt{erase}(it)$: elimina de la llista L l’element apuntat per it; retorna un iterador al successor de l’element esborrat
- Hi ha altres versions d’`insert` i `erase` però recomanem fer servir només aquestes.

Iteradors

- Llistes i iteradors “treballen” coordinadament
- Operacions que ens tornen iteradors:
  - L.`begin()`: torna un iterador apuntant al primer element de la llista L
  - L.`end()`: torna un iterador apuntant “fora”—a un element fictici successor de l’últim—de la llista L
  - Si L és buida, aleshores `L.begin() == L.end()`

Exemple d'evolució d'una llista



Iteradors

```
list<int> l;
list<string> ls;
list<int>::iterator it = l.begin();
list<int>::iterator it2 = l.end();
list<string>::iterator it3 = ls.begin();
it = it3; // error!! són de tipus diferents
```

Cada tipus d'iterador es defineix com a subclasse de la classe "contenidora"

Iteradors constants

- Iteradors constants (`const_iterator`): prohibeixen modificar l'objecte referenciat per l'iterador
- S'han d'utilitzar per a recòrrer una llista rebuda per referència constant

```
list<int>::const_iterator it, it2;
it = it2;  // OK, it és constant
++it;      // OK
v = *it;   // OK
*it = v+3; // error!!!
```

Recorreguts: Iterators o Iterators Constants

Iterator:

```
/* Pre: cert */
/* Post: A tots els elements de la llista s'els ha sumat x */
void sumar_a_llista(list<int>& L, int x) {
   list<int>::iterator it = L.begin();
   while (it != L.end()) {
      *it=(*it)+x;
      ++it;
   }
}
```

Iterator constant:

```
void imprimir_llista(const list<int>& L) {
  for(list<int>::const_iterator it=L.begin();it != L.end();++it)
     cout<<(*it);
  cout<<endl;
}
```

Especificació de la classe `list`

```
template <class T> class list {
public:
// Subclasses de la classe llista
     class iterator { ... };
     class const_iterator { ... };

// Constructores

/* Pre: cert */
/* Post: El resultat es una llista sense cap element */
list();

// Modificadores
/* Pre: cert */
/* Post: La llista implícita queda buida */
void clear();
```

Les llistes també tenen operacións push_back, push_front, pop_back i pop_front.

Especificació de la classe genèrica `Llista`

```cpp
/* Pre: it referencia algun element existent $a_i$ a la llista o
        és igual a end(), la llista és $[a_1,\ldots,a_n]$ */
/* Post: L'element s'ha inserit davant de l'element referenciat
         per it, la llista és ara $[a_1,\ldots,x,a_i,\ldots]$ */
void insert(iterator it, const T& x);

/* Pre: it referencia algun element $a_i$ existent a la
        llista $[a_1,\ldots,a_n]$, $n>0$ */
/* Post: S'ha eliminat l'element referenciat per it, la llista
         és ara $[a_1,\ldots,a_{-1},a_{i+1}\ldots,a_n]$ i torna
         un iterador al sucessor de l'element eliminat */
iterator erase(iterator it);

/* Pre: $l=[y_1,\ldots,y_m]$, $l$ i la llista implícita són
        objectes diferents, i it referencia algún element $x_i$ de
        la llista implícita $[x_1,\ldots,x_n]$ */
/* Post: La llista implícita és ara
        $[x_1,\ldots,x_{i-1},y_1,\ldots,y_m,x_i,\ldots,x_n]$ i $l$ és buida */
void splice(iterator it, list& l);
```

Especificació de la classe genèrica `Llista`

```
// Consultores

/* Pre: cert */
/* Post: torna cert si i només si la llista és buida */
bool empty() const;

/* Pre: cert */
/* Post: torna el nombre d'elements de la llista*/
int size() const;

// tornen iteradors al primer element de la llista
 const_iterator begin() const;

 iterator begin();

// tornen iteradors a l'element fictici succesor de l'últim
// de la llista
 const_iterator end() const;

 iterator end();
```

Errors freqüents en l'ús d'iteradors

- Fer servir un iterador amb una llista a la qual no està vinculat.
- Tenir l'iterador posicionat al `begin()` i retrocedir.
- Tenir l'iterador posicionat a l'`end()` i avançar.
- Tenir l'iterador posicionat a l'`end()` i consultar o modificar l'element.
- Tenir l'iterador posicionat a l'`end()` i esborrar l'element.

Sumar tots els elements d'una llista d'enters

```
/* Pre: cert */
/* Post: El resultat és la suma dels elements de l */
int suma(const list<int>& l) {
    int s = 0;
    for(list<int>::const_iterator it=l.begin();it !=l.end();++it) {
        s += *it;
    }
    return s;
}
```

Cerca senzilla en una llista d'enters

```
/* Pre: cert */
/* Post: El resultat indica si x és o no a l */
bool pertany(const list<int>& l, int x) {
    list<int>::const_iterator it = l.begin();
    while (it != l.end() and (*it != x))
      ++it;
    return it != l.end();
}
```

Dir si una llista és capicua

[4,8,5,8,4], [7], [4,8,8,4] són capicues

```
/* Pre: cert */
/* Post: El resultat diu si l es capicua */
bool capicua(const list<int>& l);
   list<int>::const_iterator it1 = l.begin();
   list<int>::const_iterator it2 = l.end();
   for (int i = 0; i < l.size()/2; ++i) {
       --it2;
       if (*it1 != *it2) return false;
       ++it1;
   }
   return true;
}
```

Recordeu que no es pot comparar `it1 < it2`.

Inserint elements ordenadament

Donada una llista l d'strings en ordre alfabètic no decreixent i un nou string s, inserir s a la llista l, mantentint l'ordre.

```
// Pre: l = L
// Post: l conté els elements d'L i s, i està en ordre
// no decreixent
void inserir_ordenadament(list<string>& l, string s);
   list<string>::iterator it = l.begin();
   while (it != l.end() and *it < s) ++it;
   // it == l.end() ó *it és un string ≥ s
   // per tant s s'ha d'inserir com a predecessor
   // de l'element apuntat per it
   l.insert(it, s);
}
```

Splice: Insert a l'engrós!

- Si `l1 = [1,2,3,4,5,6]`, `it` apunta al 4, i `l2 = [10,20,30]` llavors

  `l1.splice(it,l2),`

  queda
  `l1 = [1,2,3,10,20,30,4,5,6]`, `it` apunta a 4, i `l2` queda buida

Splice: Insert a l'engrós!

- Si `l1 = [1,2,3,4,5,6]`, `it` apunta al 4, i `l2 = [10,20,30]` llavors

  `l1.splice(it,l2),`

  queda
  `l1 = [1,2,3,10,20,30,4,5,6]`, `it` apunta a 4, i `l2` queda buida

- Per concatenar dues llistes farem:
  `l1.splice(l1.end(),l2)`

- El cost de `splice` és constant, no depèn de les longituds de les llistes

Fusió ordenada de llistes

```
/* Pre: l = L, l és una llista d'enters ordenada ascendentment
        lc és una llista d'enters ordenada ascendentment   */
/* Post: l està ordenada creixent, conté L i els elements de lc*/
void merge_llistas(list<int>& l,const list<int>& lc) {
   list<int>::iterator it = l.begin();
   list<int>::const_iterator itc = lc.begin();
   while (it != l.end() and itc != lc.end()) {
      if (*it  <= *itc) ++it;
      else if (*it > *itc) {
         l.insert(it, *itc);
         ++itc;
      }
   }
   while (itc != lc.end()){
      l.insert(it,*itc);
      ++itc;
   }
   //Si el primer bucle acaba perqué itc == lc.end()
   //no cal fer res perqué l no canvia.
}
```

En general cal un tractament final per cada llista per tractar la part no tractada en el bucle principal. En aquest cas només

Fusió ordenada de llistes

Suposem que l té n elements i que lc conté m elements.

El algorisme de “fusió ordenada” avança en cada pas/iteració en una de les dues llistes (o totes dues) i per tant el nombre d’operacions que farem serà de l’ordre de $n + m$ operacions.

Aquesta solució eficient del problema aprofita que les dues llistes estàn ordenades alfabèticament!

Funció on es crea una llista

Si volem conservar la llista original i que la llista modificada sigui nova, podem fer una funció, on haurem de fer servir l'operació `insert` per generar la llista resultat.

```
list<int> suma_llista_k(const list<int>& l, int k)
/* Pre: cert */
/* Post: Cada element del resultat és la suma de k
i l'element de l a la seva mateixa posició */
{
  list<int>::const_iterator it=l.begin();
  list<int> l2;
  list<int>::iterator it2=l2.begin();
  while (it != l.end()) {
    l2.insert(it2,*it+k);
    ++it;
  }
  return l2;
}
```

Operació per escriure una llista

La següent acció serveix per escriure els continguts d'una llista amb templates. Serveix per a llistes de qualsevol tipus.

```
template<typename T>
void printList(const list<T> &l)
{
  bool printcomma = false;
  typename list<T>::iterator it;
  for(it = l.begin();it != l.end();it++) {
    if (printcomma) cout << ",";
    printcomma = true;
    cout << *it;
  }
  cout << endl;
}
```