Estructures lineals II: Llistes

Ricard Gavaldà

Programació 2

Facultat d'Informàtica de Barcelona, UPC

Primavera 2019

Aquesta presentació no substitueix els apunts

Contingut

`push_back` en vectors

`push_back` en vectors

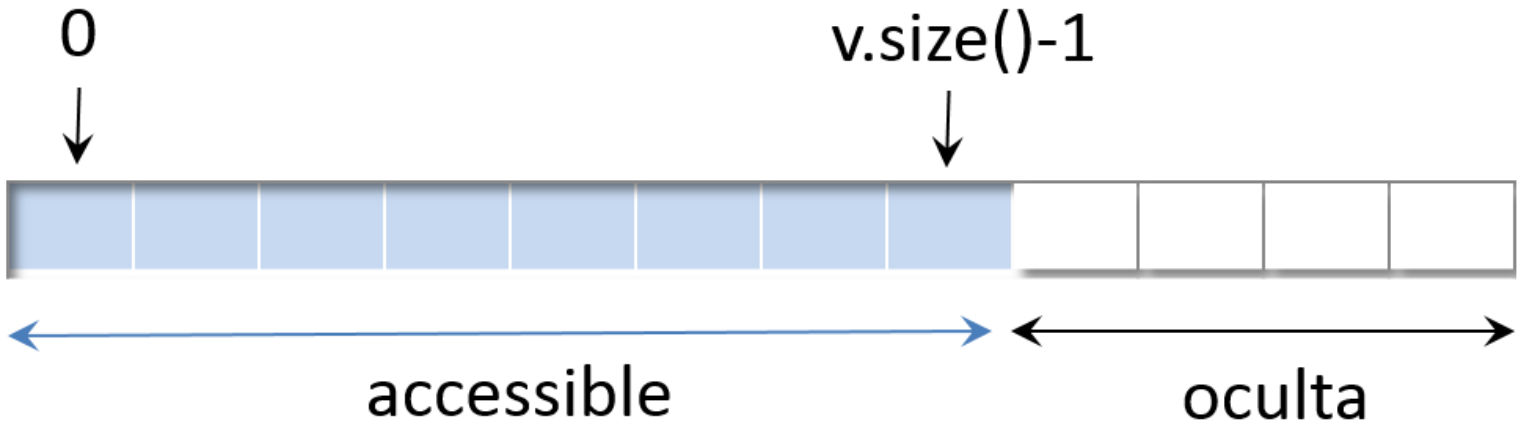
- `v.push_back(elem)` afegeix `elem` al final de `v`
- `v.size()` s'incrementa en 1

- Cal mirar si la memòria que ocuparia el nou element és lliure
- Si ja està ocupada per una altra variable, cal moure (*reallocate*) el vector:
  1. crear un vector nou (en algun altre lloc) de mida `v.size()+1`,
  2. moure els elements actuals al nou vector,
  3. afegir `elem` a la darrera posició
  4. i destruir el vector antic

`push_back` en vectors (2)

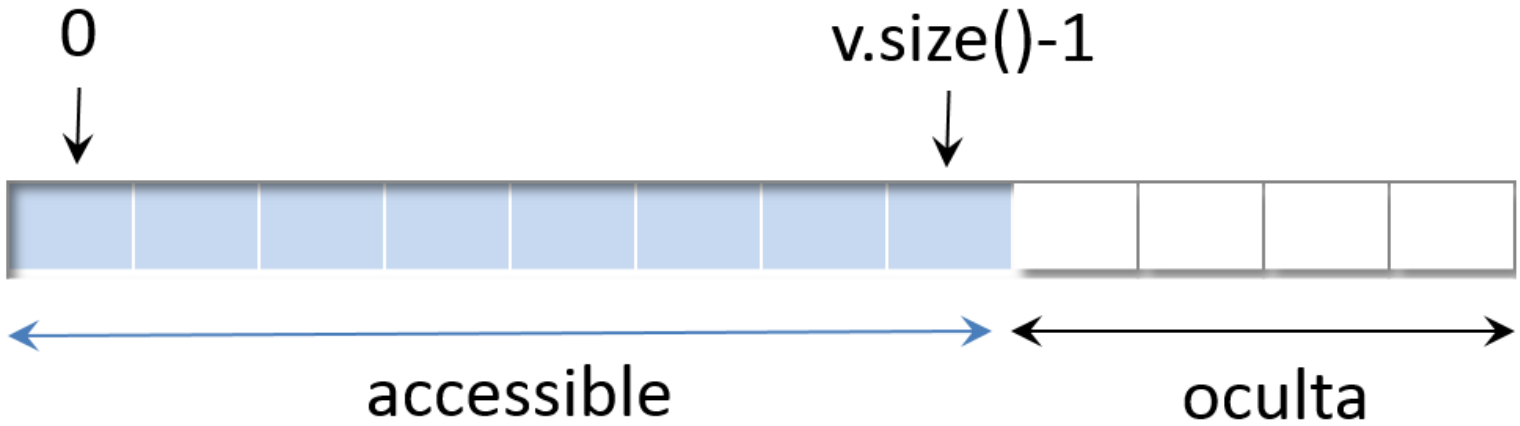
- Molt ineficient si hem de reallocatar a cada `push_back`
- $1 + 2 + 3 + 4 + \cdots + (N - 1)$ moviments per tenir un vector de mida N
- Això és $\simeq N^2/2$

`push_back` en vectors (3)



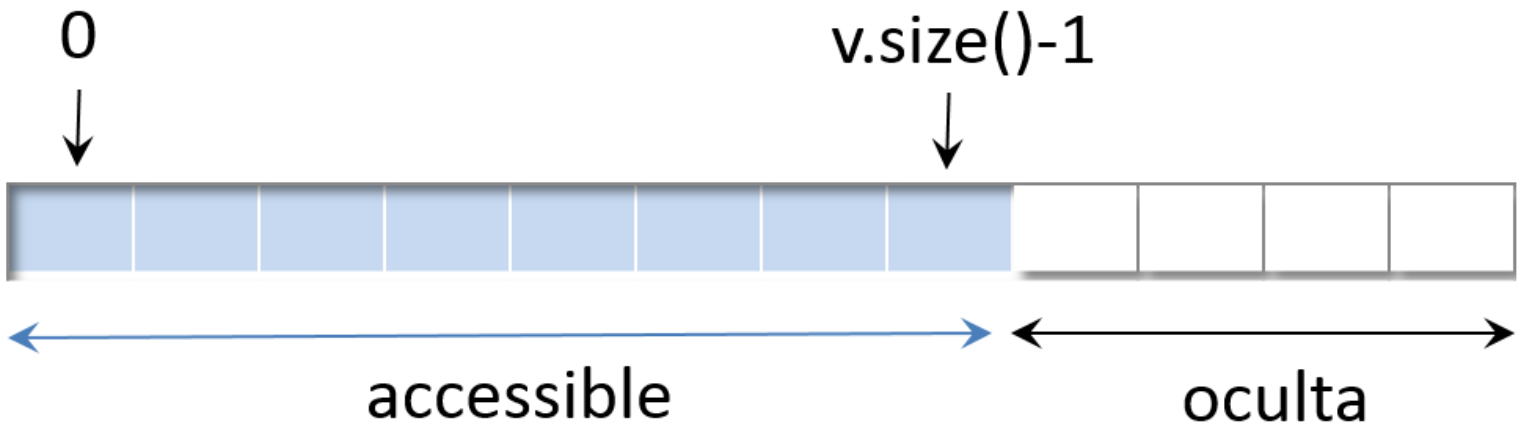
- Els vectors tenen una part oculta (espai ocupat) i una part accessible
- `v.size()` retorna la mida de la part accessible
- És un error accedir amb [...] més enllà de la part accessible

`push_back` en vectors (4)



- Si queda algun element a la part oculta, `push_back` n'ocupa el primer
- Si no, *reallocate* a un vector de mida `2*v.size()`
- La mida (accessible + oculta) és $2^i \cdot$ mida inicial.

push_back en vectors (5)



Imaginem que mida inicial és 1 per simplificar
Per tenir un vector de `size` N farem

$$1 + 2 + 4 + 8 + 16 + \cdots + 2^k = 2^{k+1} - 1$$

moviments, on $2^k < N \leq 2^{k+1}$
$\rightarrow$ com a molt $2N$ moviments
$\rightarrow$ 2 moviments en mitjana per crida a `push_back`

`push_back` en vectors (6)

Exemple de *Cost Amortitzat*, no *Cost Cas Pitjor*:

De tant en tant, alguna crida a `push_back` serà costosa, però després amortitzem la feina amb molts `push_back` barats

- En realitat, això s'implementa amb apuntadors
- Altres operacions: `pop_back()`, `resize()`, `reserve()`, `insert(i,x)`, `erase(i)`
- No les useu si no sabeu bé què esteu fent
- `insert(i)`, `erase(i)`: Cost proporcional a `v.size()`

Inserció en vectors

La classe `vector` té un mètode `insert(idx,valor)`
P.ex. si `v == ['a','b','c']` i fem `v.insert(1,'e')`, tenim `v == ['a','e','b','c']`

Té cost proporcional a `v.size()`

`push_back(x)` és una implementació eficient de `insert(v.size(),x)`

Dues versions d'un problema

Llegir una seqüència d'int's i deixar-los en un vector en ordre invers al d'arribada

Versió 1:

```
vector<int> v;
while (cin >> x) v.insert(0,x);
```

Versió 2:

```
vector<int> v;
while (cin >> x) v.push_back(x);
for (int i = 0; i < v.size()/2; ++i)
    swap(v[i],v[v.size()-i-1]);
```

Quina és millor?

Llistes

Introducció a les llistes. Contenidors i iteradors

contenidor: estructura de dades per emmagatzemar objectes

Són `template`: necessiten un tipus com a paràmetre

S'usen amb *iteradors:* classe per desplaçar-nos pel contenidor

Avui: llista (`list`)

Propietats de les llistes, respecte vectors

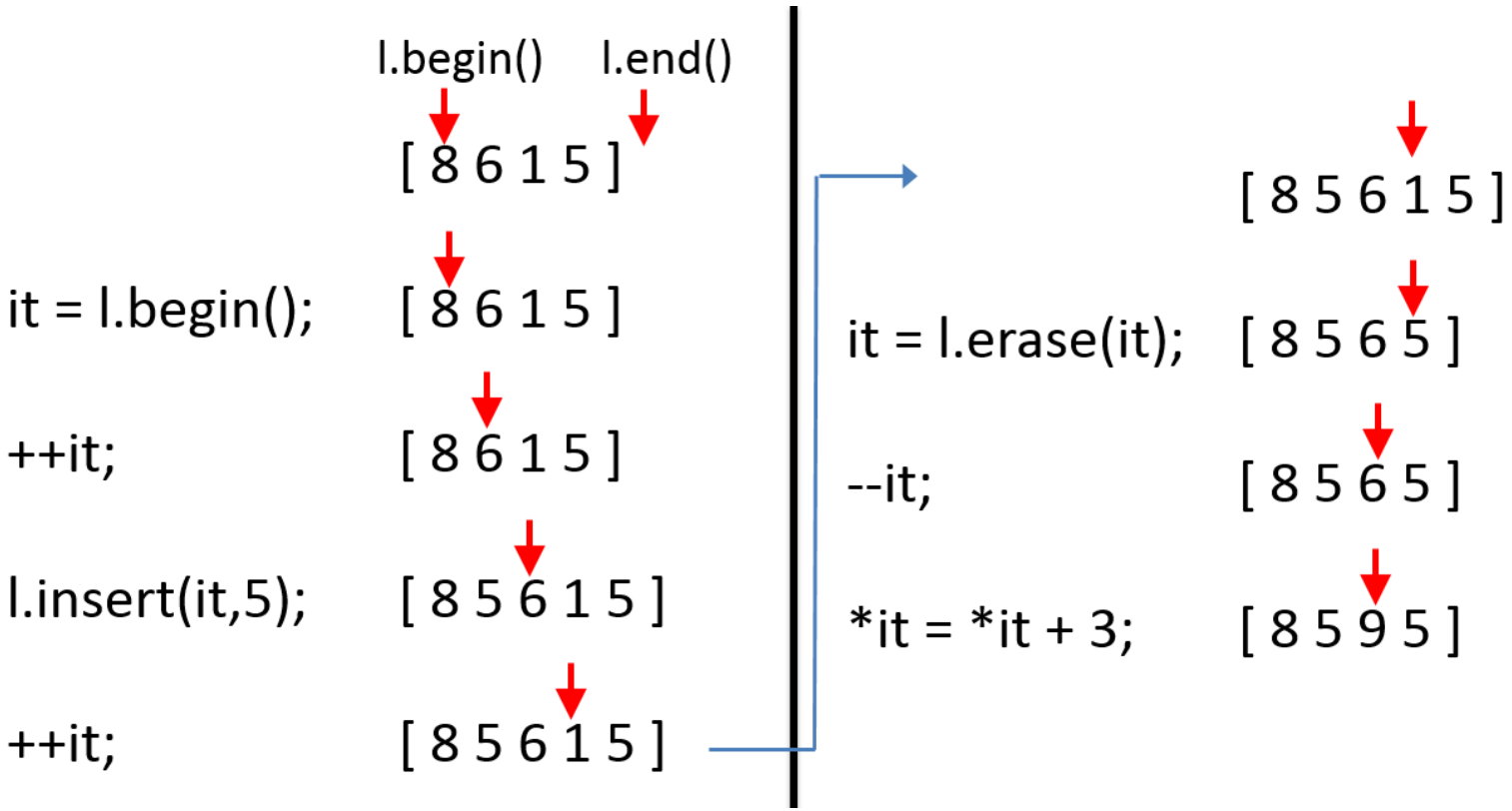
Cons:

- Accés directe (donat i, accedir a l'element i-èssim) costós

Pros:

- Recorreguts seqüencials tan eficients com en vectors
- Inserir element al punt on "som" eficient
- Esborrar element al punt on "som" eficient
- Concatenar llistes eficient

Exemple d'evolució d'una llista



Iteradors: declaració (instanciació)

Mètode `begin()`: retorna un iterador que referencia el primer element del contenidor, si és que existeix

Mètode `end()`: retorna un iterador que referencia un element fictici, posterior al darrer element

```
list<Estudiant>::iterator it = l.begin();
list<Estudiant>::iterator it2 = l.end();
```

"::": La definició de l'iterador pertany al contenidor
Si una llista `l` és buida, aleshores `l.begin()` = `l.end()`

Iteradors: operacions amb iteradors

Sí:

- `it1 = it2;`
- `it1 == it2`, `it1 != it2`
- `*it` (excepte si `it == l.end()`)
- `++it` , `--it` (excepte si a `l.end()` i `l.begin()`)

NO:

- `it = it + 5; it = it - 1;`
- `if (it1 < it2) ...`

Els iteradors no són `int`

++ i −− : "avança" i "retrocedeix", no "incrementa" i "decrementa"

Iteradors: operacions amb iteradors

Esquema freqüent:

```
list<T> l;
...
list<T>::iterator it = l.begin();
while (it != l.end() and not (condicio sobre *it)) {
   ... accedir a *it ...
   ++it;
}
```

Iteradors constants

Iteradors constants: prohibeixen modificar l'objecte referenciat per l'iterador

```
list<Estudiant>::const_iterator it;
```

Sí:

- `it = it2`
- `++it`
- `v = *it`

NO:

- `*it = ...`

`*it` és una constant. `it` no és una constant

Especificació de la classe genèrica `Llista`

Vegeu Llista.doc

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();
}
```

Exercici: cerca en una llista d'estudiants

```
/* Pre: cert */
/* Post: El resultat ens indica si hi ha algun estudiant
   amb dni x a l o no */
bool pertany(const list<Estudiant>& l, int x);
```

Modificar una llista sumant un valor k a tots els elements

```
/* Pre: cert */
/* Post: Cada element de l és com el de la mateixa posició
   de la l original, més k */
void suma_k(list<int>& l, int k) {
    list<int>::iterator it = l.begin();
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
```

Alternativa

Alternativa (pitjor) a

```
*it += k;
++it;
```

seria

```
int aux = (*it) + k;
it = l.erase(it);
l.insert(it,aux);
// sense ++it!
```

Un iterador està lligat a un element, no a una posició de la llista

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);
```

Dir si una llista és 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;
}
```

Exercici: Penseu com fer-ho sense `l.size()`.
Cada element s'ha de consultar un cop com a molt.
Recordeu que no es pot comparar $it1 < it2$

Splice: Insert a l'engrós!

Si l1 = [1,2,3,4,5,6], it apunta a 4, l2 = [10,20,30]

i fem l1.splice(it,l2), queda

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

Nota: Concatenar és l1.splice(l1.end(),l2)

Nota: La STL de C++ té versions més complexes de splice

A recordar per a l'examen:

- On inserten `insert` i `splice`
- On queda l'iterador després d'`erase`, `insert` i `splice`

Vectors vs. llistes

- Recorregut seqüencial: Temps lineal en els dos
  - constant per element
- Accés directe a i-èssim: Constant en vectors, temps i en llistes
- Inserir un element
  - Constant en llistes
  - En vectors, afegir al final és constant en mitjana (`push_back()`)
  - En vectors, inserir pel mig és costós
- Esborrar un element: constant en llistes, costós en vectors
- Splice: constant en llistes, costós en vectors

Accés directe?

Accés directe per posició en llistes. Si cal ...

```
// pre: 0 <= i < l.size()
// post: retorna l'i-essim element de l
double get(const list<double>& l, int i) {

    list<double>::const_iterator it = l.begin();
    for (int j = 0; j < i; ++j) ++it;
    return *it;

}
```

Accés directe

```
list<double>::iterator it = l.begin();
double sum = 0;
while (it != l.end()) { sum += *it; ++it; }
```

vs.

```
double sum = 0;
for (int i = 0; i < l.size(); ++i) sum += get(l,i);
```

Accés directe

Si $L =$ `l.size()`

Solució 1: Temps $= L$

Solució 2: Temps $= 1 + 2 + \cdots + L \simeq L^2/2$

Per a $L = 100.000$,

Solució 1: 100.000 Solució 2: 5.000.000.000

[http://www.joelonsoftware.com/articles/fog0000000319.html]

Recordeu: Iterators are your friends