Estructures lineals

Assignatura PRO2

Setembre 2017

Índex

3 Estructures de dades lineals **7**
3.1 Especificació de la classe genèrica Pila o *stack* 7
3.2 Especificació de la classe genèrica Cua o *queue* 9
3.3 Ús de les classes Pila i Cua 11
3.3.1 Expressions en notació postfixa 11
3.3.2 Caixes de supermercat 13
3.4 Llistes i iteradors, *list*s i *iterator*s 14
3.5 Especificació de la classe genèrica Llista 15
3.6 Operacions de recorregut de llistes 17
3.7 Operacions de cerca en llistes 19
3.8 Modificació i generació d'una llista 20

Tipus genèrics o parametritzats de dades

El mecanisme de *templates* de C++ permet fer servir tipus de dades com a paràmetres. D'aquesta forma, un tipus de dades pot aparèixer com a paràmetre en la definició d'una classe o mètode. Es pot, per tant, treballar amb estructures de dades genèriques, on podem guardar qualsevol tipus d'element, donat que aquest tipus és un paràmetre. En el moment de crear l'estructura només caldrà *instanciar* el tipus d'element que hi guardarem.

En aquest capítol no implementarem estructures de dades genèriques, sino que farem servir estructures ja implementades. Per fer servir correctament aquestes estructures ens cal conèixer una sèrie de recomanacions i saber com fer la instanciació.

Els vectors són un exemple d'estructura de dades genèrica que ja coneixem. Quan declarem

```
vector<int> v;
```

estem creant una instància del tipus genèric `vector<T>` amb `T = int`.

Normalment, per a les estructures genèriques que farem servir als nostres programes, i també pels objectes continguts dintre de les estructures, podrem fer servir l'operador d'assignació = per fer còpies. En els casos en que no sigui així es farà constar i es proporcionarà algun mètode alternatiu per fer còpies.

Una conseqüència de treballar amb estructures genèriques és que no pot ser genèric cap mètode que depengui de la classe dels components. Cap mètode genèric pot tenir en el seu codi ni lectures, ni escriptures dels elements, perquè encara no se sap a quina classe pertanyeran aquests elements. Això obliga a crear operacions fora de la classe genèrica per fer les operacions de lectura i escriptura d'estructures que continguin elements d'una classe concreta. Per cada classe d'elements, haurem d'implementar operacions de lectura i escriptura per l'estructura de dades contenidora. Aquestes operacions no són de la classe i no tenen paràmetre implícit.

Capítol 3

Estructures de dades lineals

Una estructura de dades és lineal si els seus elements formen una seqüència. Podem representar l'estructura com

$$e_1, e_2, \cdots, e_n$$

on $n \geq 0$. Si $n = 0$, l'estructura està buida. Si $n = 1$, l'estructura té un element que és al mateix temps el primer i l'últim, i no té ni antecessor ni successor. Si $n = 2$, hi ha dos elements, el primer element, sense antecessor i que té com a successor el segon, i el segon element, sense successor i que té com a antecessor el primer. Si $n > 2$, tots els elements a partir del segon fins al penúltim tenen antecessor i successor.

A classe de teoria i laboratori es presentaran les estructures lineals pila (stack), cua (queue) i llista (list). Les diferents estructures de dades lineals es diferencien per com es pot accedir als seus elements.

3.1 Especificació de la classe genèrica Pila o *stack*

Una pila és una estructura lineal de dades molt usada en programació. Es diu així perquè és similar a una pila normal, per exemple de plats, on quan afegim un plat a la pila el posem a dalt de tot i quan traguem un plat de la pila agafem el darrer que s'ha posat, és a dir, el que està a dalt de tot. És una estructura de tipus *LIFO*, que en anglès és un acrònim de *Last in, first out*. El nom estàndard d'aquesta estructura en C++ i en anglès és *stack*. Farem servir el nom *pila* en el text d'aquests apunts quan parlem de piles en general i `stack` als programes d'exemple. Els noms de variables pila seran normalment `p`, `q`, etc.

La classe `stack` de C++ de fet és només una restricció d'una altra classe ja existent que és més potent i versàtil. L'especificació que donem seguidament està adaptada de l'implementació estàndard de la classe `stack` on s'han omès detalls que en aquest punt no tenen importància. Observeu l'aparició del terme `template` a l'especificació de la classe `stack`. El nom `T` fa referència al tipus dels elements que contindrà la pila, que pot ser qualsevol. Per fer servir la pila hem de indicar quin tipus d'elements contindrà i allà on hi hagi un paràmetre `T`, posar una expressió del tipus triat.

```
template <class T> class stack{
// Tipus de mòdul: dades
// Descripció del tipus: Estructura lineal que conté elements de tipus T i que
// permet consultar i eliminar només l'últim element afegit

private:

public:

// Constructores

stack();
/* Pre: cert */
/* Post: El resultat és una pila sense cap element */

stack(const stack & original);
/* Pre: cert */
/* Post: El resultat és una còpia d'original */

// Modificadores

void push(const T& x);
/* Pre: cert */
/* Post: El paràmetre implícit és com el paràmetre implícit original amb x afegit
   com a darrer element */

void pop();
/* Pre:  El paràmetre implícit no està buit */
/* Post: El paràmetre implícit és com el paràmetre implícit original però sense el
   darrer element afegit al paràmetre implícit original */

// Consultores

T top() const;
/* Pre: El paràmetre implícit no està buit */
/* Post: El resultat és el darrer valor afegit al paràmetre implícit */

bool empty() const;
/* Pre: cert */
/* Post: El resultat indica si el paràmetre implícit és buit o no */

int size() const;
/* Pre: cert */
/* Post: El resultat és el nombre d'elements del paràmetre implícit */
};
```

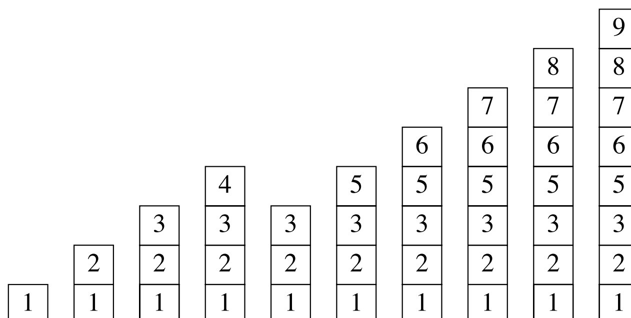

Figura 3.1: Exemple d'evolució d'una pila

Per instanciar piles d'un tipus concret escriurem `stack<tipus> nom_pila;`, tal i com es feia amb els vectors. Per exemple `stack<int> p;` o `stack<Estudiant> q;`, etc.

A la figura 3.1 veiem un exemple d'evolució d'una pila d'enters. Afegim successivament els valors 1, 2, 3, i 4; traiem l'únic valor accessible, i afegim successivament els valors 5, 6, 7, 8 i 9.

3.2 Especificació de la classe genèrica Cua o *queue*

Una cua és una estructura lineal de dades molt usada en programació. Es diu així perquè és similar a una cua de gent a la vida real. Quan s'afegeix un element a la cua, es posa darrera de tots els elements que hi havia, i quan es treu un element de la cua s'agafa el primer que ha arribat, és a dir, el que porta més temps a la cua. És una estructura de tipus *FIFO*, que en anglès és un acrònim de *First in, first out*.

El nom estàndard d'aquesta estructura en C++ i en anglès és *queue*. Farem servir el nom *cua* en el text d'aquests apunts quan parlem de cues en general i `queue` als programes d'exemple. Els noms de variables cua seran normalment `c`, `c1`, `c2`, etc.

La classe `queue` de C++ de fet és només una restricció d'una altra classe ja existent que és més potent i versàtil. L'especificació que donem seguidament està adaptada de l'implementació estàndard de la classe `queue` on s'han omès detalls que en aquest punt no tenen importància. El rol del terme `template` i el tipus `T` a l'especificació de la classe `queue` és el mateix que el que vam comentar pel cas de la classe `stack`.

```
template <class T> class queue{
// Tipus de mòdul: dades
// Descripció del tipus: Estructura lineal que conté elements de tipus T i que
// permet consultar i eliminar només el primer element afegit

private:

public:
```

```
// Constructores

queue();
/* Pre: cert */
/* Post: El resultat és una cua sense cap element */

queue(const queue &original);
/* Pre: cert */
/* Post: El resultat és una cua còpia d'original */

// Modificadores

void push(const T& x);
/* Pre: cert */
/* Post: El paràmetre implícit és com el paràmetre implícit original amb x afegit
   com a darrer element */

void pop();
/* Pre:  El paràmetre implícit no està buit */
/* Post: El paràmetre implícit és com el paràmetre implícit original però sense el
   primer element afegit al paràmetre implícit original */

// Consultores

T front() const;
/* Pre: El paràmetre implícit no està buit */
/* Post: El resultat és el valor més antic afegit al paràmetre implícit */

bool empty() const;
/* Pre: cert */
/* Post: El resultat indica si el paràmetre implícit és buit o no */

int size() const;
/* Pre: cert */
/* Post: El resultat és el nombre d'elements del paràmetre implícit */
};
```

Per instanciar cues d'un tipus concret escriurem `queue<tipus> nom_cua;`, per exemple escriurem `queue<int> c;` o `queue<Estudiant> q;`, etc.

A la figura 3.2 veiem un exemple d'evolució d'una cua d'enters. Els valors 1, 2 i 3 demanen torn, després s'avança, els valors 4 i 5 demanen torn, es torna a avançar i els valors 6 i 7 demanen torn.

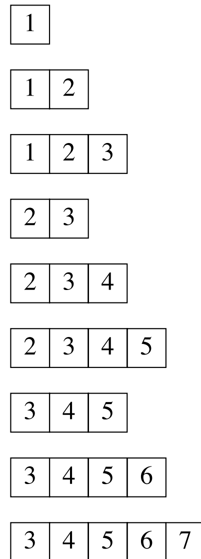

Figura 3.2: Exemple d'evolució d'una cua

3.3 Ús de les classes Pila i Cua

3.3.1 Expressions en notació postfixa

Els compiladors comproven si els programes contenen errors sintàctics. Normalment, l'omissió d'un símbol, per exemple un parèntesi o el marcador de l'inici d'un comentari, pot generar molts missatges de diagnòstic del compilador que no serveixen per identificar el veritable error que conté el programa. Una eina útil en aquesta situació és un programa que comprovi que tot parèntesi, clau, claudàtor o marcador d'inici d'un bloc obert es tanca adientment. Per exemple, la seqüència [()] és correcta, pero la seqüència [(]) no ho és. Es poden trobar diverses solucions per a aquest problema on l'ús d'una pila resulta recomanable. Veurem un altre exemple a continuació.

Avaluació d'una expressió en notació postfixa

L'avaluació d'expressions aritmètiques en notació infixa requereix tenir en compte les diferents prioritats dels operadors multiplicatius i additius, les regles d'associativitat d'aquests operadors, i l'ordre d'avaluació d'arguments implicat per l'ús de parèntesis. Avaluar una expressió aritmètica en notació postfixa (també coneguda com a notació polonesa inversa) és més senzill, perquè no cal conèixer les regles de precedència entre els diferents tipus d'operadors aritmètics.

El següent programa fa servir una pila d'enters per avaluar una expressió aritmètica representada en notació postfixa. Només cal recórrer un cop la seqüència de símbols que representa l'expressió aritmètica. A mesura que es van processant aquests símbols, es fa servir la pila d'enters per emmagatzemar resultats intermitjos i per recuperar-los quan cal que siguin usats com arguments d'operadors que es troben en posicions posteriors de l'expressió aritmètica.

```
int eval(const string& expr) {
  /* Pre: expr es una expressió aritmètica en notació postfixa formada per naturals
     entre 0 i 9, i els operadors + i *; pot contenir blancs */
  /* Post: retorna el resultat d'avaluar expr */

  stack<int> arg;
  for (int i = 0; i < expr.size(); ++i) {
    if ('0' <= expr[i] and expr[i] <= '9') {
      arg.push(expr[i] - '0');
    }
    else if  (expr[i] == '*' or expr[i] == '+')
      {
        int arg1 = arg.top();
        arg.pop();
        int arg2 = arg.top();
        arg.pop();
        int result;
        if (expr[i] == '*') result = arg1*arg2;
        else result = arg1+arg2;
        arg.push(result);
      }
  }
  return arg.top();
}
```

Transformació de notació infixa a postfixa

També es pot fer servir una pila per convertir una expressió aritmètica en notació infixa en una expressió aritmètica equivalent en notació postfixa. A continuació presentem un programa que fa servir una pila de caràcters per realitzar aquesta conversió quan les expressions només contenen nombres naturals, parèntesis, els operadors aritmètics + y *, i cada subexpressió de la forma A operador B està envoltada per un parell de parèntesis, és a dir, es representa com (A operador B).

```
string to_postfix(const string& infix) {
  /* Pre: infix es una expressió aritmètica en notació infixa formada per
     valors entre 0 i 9 i els operadors + i *, sense blancs;
     cada operador s'aplica fent servir un parell de parèntesis */
     /* Post El resultat es la expressió equivalent a infix en notació postfixa */

  string postfix;
  stack<char> oper;
  for (int i = 0; i < infix.size(); ++i) {
    if (infix[i] == ')') {
      postfix += " ";  postfix += oper.top(); // concatenació
      oper.pop();
```

```
    }
    else if (infix[i] == '*' or infix[i] == '+') {
      oper.push(infix[i]);
    }
    else if ('0' <= infix[i] and infix[i] <= '9') {
      postfix += " "; postfix += infix[i];
    }
  }
  return postfix;
}
```

Per transformar ((3+5)*(2+4)) en 3 5 + 2 4 + *, ignorem els caràcters corresponents als parèntesis oberts, escrivim (3+5) en forma postfixa, emmagatzemem l'operador * a la pila, escrivim (2+4) en notació postfixa, i quan trobem el caràcter corresponent al darrer parèntesi tancat, treiem l'operador de la pila y l'escrivim.

3.3.2 Caixes de supermercat

Considerem un supermercat amb $m > 0$ caixes obertes, indexades amb els valors $1 \ldots m$. Volem simular les operacions d'afegir clients de les caixes i treure el client més antic d'una caixa. Un client té un identificador i una càrrega, tots dos enters (la càrrega és sempre positiva). Un nou client s'afegeix sempre a la caixa menys "carregada". En cas d'empat, es tria la caixa amb l'índex més petit. La càrrega d'una caixa és la suma de les càrregues dels seus clients.

Representem les caixes i els seus clients, en ordre d'incorporació, amb un vector de cues de parells d'enters. Un parell s'encapsula a la classe `ParInt`, on el primer element és l'identificador i el segon és la càrrega. També fem servir un vector d'enters per gestionar eficientment les càrregues de les caixes i un enter en $1 \ldots m$ que ens dirà en tot moment quina és la caixa amb la mínima càrrega.

Podem implementar l'operació d'afegir un client així

```
void afegeix_client(vector<queue<ParInt>>& vcaix, vector<int>& vcarg, const ParInt& p, int& min_carg)
/* Pre: cert */
/* Post: el client p s'ha incorporat a la caixa menys carregada de vcaix;
   vcarg i min_carg s'han actualitzat */
{
  vcarg[min_carg-1] += p.segon();
  vcaix[min_carg-1].push(p);
  recalcular_min_carrega_afegeix(vcarg,min_carg);
}
```

Podem implementar l'operació de treure el client més antic d'una caixa així

```
void treure_client(vector<queue<ParInt>>& vcaix, vector<int>& vcarg, int i, int& min_carg)
/* Pre: 1<=i<=vcaix.size() */
/* Post: s'ha tret el client més antic de la caixa i; vcarg i min_carg s'han actualitzat */
{
```

```
  if (not vcaix[i-1].empty()){
    vcarg[i-1] -= vcaix[i-1].front().segon();
    vcaix[i-1].pop();
    recalcular_min_carrega_treure(vcarg,min_carg,i);
  }
}
```

Noteu que a les especificacions queda implícita la correcta relació entre `vcaix`, `vcarg` i `min_carg`. Noteu també que quan un client es treu podem actualizar eficientment `min_carg`; no així quan un client s'afegeix, per això introduïm dues operacions auxiliars diferents, que s'haurien d'implementar com a exercici.

Hi ha més exemples al document "Piles i cues: més exemples"

3.4 Llistes i iteradors, *list*s i *iterator*s

En C++ direm *contenidor* a una estructura de dades on emmagatzemarem objectes d'una forma determinada. En general, els contenidors son classes genèriques que s'instancien de la manera que ja hem vist. Un exemple de contenidor que ara presentarem és la llista.

Una *llista* és una estructura de dades lineal que permet accedir a qualsevol element *sense treure els anteriors*. Això es fa mitjançant iteradors.

Un *iterador* ens permet desplaçar-nos per un contenidor i fer referència als seus elements. Depenent de les característiques del contenidor es podran fer servir diferents tipus d'iteradors. Pel cas de llistes d'enters, declarem una llista i un iterador així

```
list<int> l;
list<int>::iterator it;
```

Noteu que l'iterador no s'associa a la llista *l* sinó a la classe. Això comporta que es podria reutilitzar el mateix iterador amb diferent llistes.

Les llistes ofereixen un mètode `begin()` que retorna un iterador que referencia el primer element del contenidor, si és que existeix. Per exemple, podem fer que l'iterador anterior referenciï el primer element de la llista de la següent forma:

```
it = l.begin();
```

També hi ha iteradors constants que impedeixen modificar l'objecte referenciat per l'iterador. És obligatori fer servir iteradors constants si són per explorar una llista que sigui un paràmetre `const`. Si, per exemple, necessitem una llista d'`Estudiant` amb un iterador d'aquesta mena, cal fer:

```
list<Estudiant> l_est;
list<Estudiant>::const_iterator it2=l_est.begin();
```

`l.begin()`	Retorna un iterador que referencia el primer element d'l
`l.end()`	Retorna un iterador que referencia un element fictici posterior al darrer d'l
`++it`	Avança al següent element
`--it`	Retrocedeix a l'anterior element
`*it`	Designa l'element referenciat per `it`
`it1=it2`	Assigna l'iterador `it2` a `it1`
`it1==it2`	Diu si els iteradors `it1` i `it2` són iguals o no
`it1!=it2`	Diu si els iteradors `it1` i `it2` són diferents o no

Figura 3.3: Resum de les operacions amb iteradors

Amb l'iterador `it2` podem moure'ns per la llista `l_est` i consultar el valor dels seus elements, però no modificar-los.

Podem fer que un iterador es mogui per una llista amb la notació `++it` i `--it`, que ens permet anar al següent element i al anterior respectivament. També existeixen aquests operadors en forma postfixa.

Podem desreferenciar un iterador, obtenint l'objecte que referencia, amb la notació `*`. Per exemple, si l'iterador `it2` definit anteriorment ja s'ha mogut per `l_est`, l'element referenciat per `it2` és `*it2`, i per consultar el seu dni, podrem fer `(*it2).consultar_DNI()`. No podrem, però, modificar `*it2`, ja que `it2` ha estat definit com a constant.

Les llistes també ofereixen un mètode `end` que retorna un iterador que referencia a un element inexistent posterior al darrer element de la llista. No és recomanable desreferenciar aquest iterador.

Es poden assignar iteradors amb `it1=it2;` i comparar-los per veure si són iguals o no (és a dir, si referencien al mateix element), amb `it1==it2` i `it1!=it2` respectivament. Un cas particularment útil per recórrer llistes és comparar l'iterador amb el que ens movem amb l'iterador retornat pel mètode `end` que ens indica el final de l'estructura que estem tractant, per exemple fent `(it != l.end())` on `l` és una llista amb un iterador `it`. A la figura 3.3 hi ha un resum de les operacions amb iteradors.

3.5 Especificació de la classe genèrica Llista

Com en el cas de l'estructures anteriors, només donem una part de l'especificació de la classe *list*. Hi ha altres operacions que no mencionem i algunes de les mencionades tenen altres variants que no farem servir i que per tant no apareixen en aquests apunts. Per exemple, només hi ha una versió de les operacions `insert` o `splice`. Aquesta darrera la farem servir bàsicament per concatenar llistes.

```
template <class T> class list {

// Tipus de mòdul : dades
```

```
// Descripció del tipus: Estructura lineal que conté elements de tipus T, que es
// pot començar a consultar pels extrems, on des de cada element es pot accedir
// a l'element anterior i posterior (si existeixen), i que admet afegir-hi
// i esborrar-hi elements a qualsevol punt

private:

public:

// Constructores

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

list(const list & original);
/* Pre: cert */
/* Post: El resultat és una llista còpia d'original */

// Modificadores

void clear();
/* Pre: cert */
/* Post: El paràmetre implícit és una llista buida */

void insert(iterator it, const T& x);
/* Pre: it referencia algun element existent al paràmetre implícit o
   és igual a l'end() d'aquest */
/* Post: El paràmetre implícit és com el paràmetre implícit original amb x
   davant de l'element referenciat per it al paràmetre implícit original */

iterator erase(iterator it);
/* Pre: it referencia algun element existent al paràmetre implícit,
   que no és buit */
/* Post: El paràmetre implícit és com el paràmetre implícit original sense
   l'element referenciat per l'it original; el resultat referencia l'element
   següent al que referenciava it al paràmetre implícit original */

void splice(iterator it, list& l);
/* Pre: l=L, it referencia algun element del paràmetre implícit o
   és igual a l'end() d'aquest; el p.i. i l no són el mateix objecte */
/* Post:  El paràmetre implícit té els seus elements originals i els
   d'l inserits davant de l'element referenciat per it; l està buida */

// Consultores
```

```
bool empty() const;
/* Pre: cert */
/* Post: El resultat indica si el paràmetre implícit té elements o no */

int size() const;
/* Pre: cert */
/* Post: El resultat és el nombre d'elements del paràmetre implícit */
};
```

En aquesta llista s'haurien d'afegir les operacions de iteradors. Noteu que no hi ha una operació de consulta d'elements perquè farem servir els iteradors per desreferenciar elements. També cal tenir en compte que si s'elimina el darrer element d'una llista, l'iterador referenciarà l'`end()`. A la figura 3.4 veiem un exemple de com evoluciona una llista després d'aplicar diferents operacions. Hem afegit l'operació `splice` que no farem servir gaire sovint, però que ens pot ser útil per concatenar dues llistes, per exemple `l1` i `l2`. Per aconseguir-ho haurem de escriure `l1.splice(l1.end(),l2);`. Si consultem l'especificació, veurem que així `l2` s'insereix a partir del final de `l1`.

Com passava amb les classes `stack` i `queue`, si un programa que fa servir la classe `list` executa operacions incorrectes, pot passar que el programa es continui executant, treballant de forma imprevisible. Per evitar aquesta mena de problemes cal, a l'hora de compilar, fer servir el *flag* `D_GLIBCXX_DEBUG`, és a dir:

```
> g++ -c elmeuprograma.cc -D_GLIBCXX_DEBUG -I$INCLUSIONS
```

i enllaçar normalment. Seguint aquestes instruccions, en el moment que el programa intenti accedir incorrectament a una llista, el programa s'interromprà donant un missatge d'error. D'aquesta manera ens serà més fàcil localitzar on s'ha produït l'error i corregir-lo. Entre els errors durant l'execució que ara fan interrompre el programa amb un missatge d'error explicatiu hi ha:

- Fer retrocedir un iterador a l'element `begin()` d'una llista, és a dir, fer `--it` quan per alguna llista `L` es compleix `it==L.begin()`.
- Fer avançar un iterador a l'element `end()` d'una llista, és a dir, fer `++it` quan per alguna llista `L` es compleix `it==L.end()`.
- Desreferenciar un iterador a l'element `end()` d'una llista
- Desreferenciar un iterador no assignat

3.6 Operacions de recorregut de llistes

Oferim un exemple d'algorisme d'exploració: sumar tots els elements d'una llista d'enters. Passem la llista per referència constant perquè així estalviem que es faci una còpia del paràmetre per l'operació. Contràriament al que passa amb piles i cues, es pot explorar una llista sense modificar-la. Si la llista es passa per referència constant tots els seus iteradors han de ser també iteradors constants.

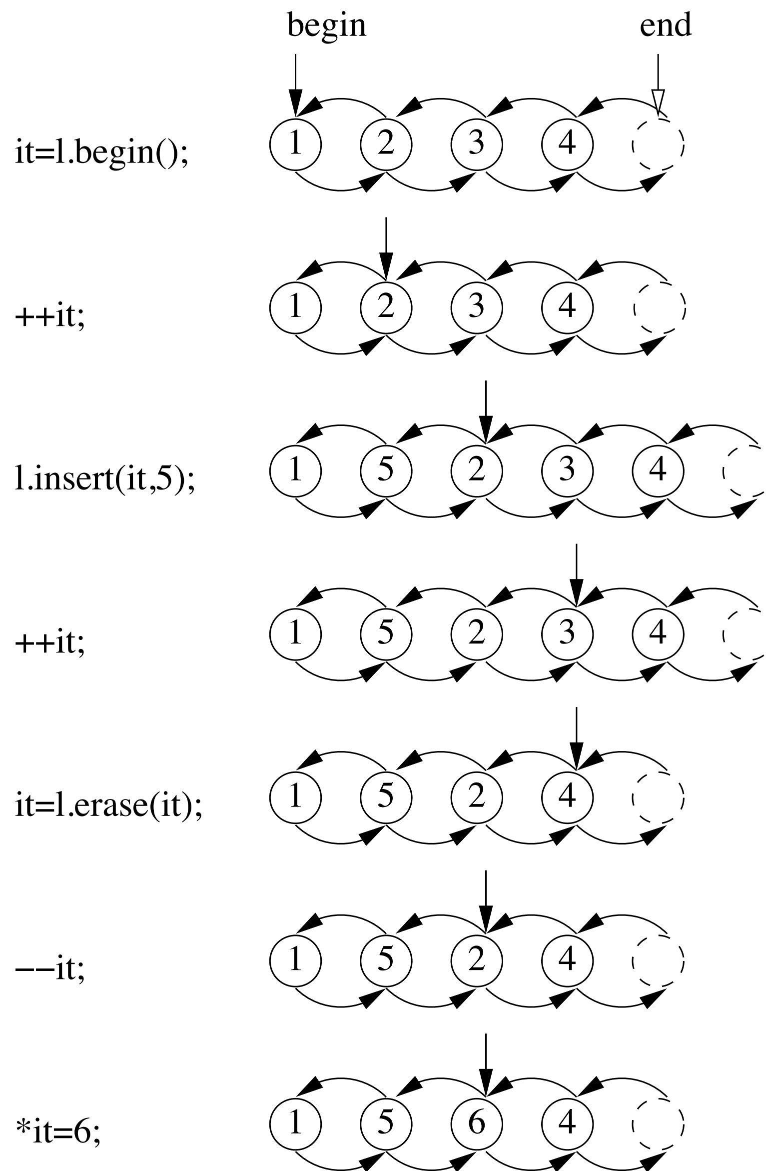

Figura 3.4: Exemple d'evolució d'una llista

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

Noteu que si una llista `l` és buida, llavors `l.begin()` i `l.end()` són iguals.

3.7 Operacions de cerca en llistes

Oferim una versió d'una cerca senzilla en una llista d'enters.

```
bool pert_llista_int(const list<int>& l, int x)
/* Pre: cert */
/* Post: El resultat indica si x hi és o no a l */
{
  bool b = false;
  list<int>::const_iterator it= l.begin();
  while ( it != l.end() and not b){
    if (*it == x) b= true;
    else ++it;
  }
  return b;
}
```

Repetim l'exemple per llistes d'`Estudiant`. Així veiem com treballar quan la llista conté objectes en comptes de tipus simples.

```
bool pert_llista_Estudiant(const list<Estudiant>& l, int x)
/* Pre: cert */
/* Post: El resultat ens indica si hi ha algun estudiant amb dni x a l o no */
{
  bool b =false;
  list<Estudiant>::const_iterator it = l.begin();
  while (it != l.end() and not b){
    if ((*it).consultar_DNI() == x) b= true;
    else ++it;
  }
  return b;
}
```

3.8 Modificació i generació d'una llista

Primer modifiquem una llista sumant un valor `k` a tots els elements.

```
void suma_llista_k(list<int>& l, int k)
/* Pre: l=L */
/* Post: El valor de cada element d'l és el valor corresponent a L més k */
{
  list<int>::iterator it= l.begin();
  while (it != l.end()){
    *it+=k;
    ++it;
  }
}
```

També seria vàlida aquesta solució, encara que no és preferible ja que "mou" més informació.

```
void suma_llista_k(list<int>& l, int k)
/* Pre: l=L */
/* Post: El valor de cada element d'l és el valor corresponent a L més k */
{
  list<int>::iterator it= l.begin();
  while (it != l.end()){
    int aux =  (*it) + k;
    it=l.erase(it);
    l.insert(it,aux);
  }
}
```

Si volem conservar la llista original i que la llista modificada sigui nova, podem obtenir una solució similar en forma de 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 d'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;
}
```

Noteu que per crear o modificar una llista no podem fer servir iteradors constants. També és destacable el fet que l'iterador `it2` sempre val `l2.end()`, ja que cada nou element a `l2` s'afegeix davant seu.

Per últim, si obtenim la nova llista amb una acció, ens estalviarem l'assignació al resultat quan fem la crida corresponent

```
void suma_llista_k(const list<int>& l, list<int> &l2, int k)
/* Pre: l2 és buida */
/* Post: Cada element de l2 és la suma de k i l'element d'l a la seva
   mateixa posició */
{
  list<int>::const_iterator it=l.begin();
  list<int>::iterator it2=l2.begin();

  while (it != l.end()) {
    l2.insert(it2,*it+k);
    ++it;
  }
}
```

En resum, si no volem conservar la llista original, la versió preferible és la primera, que és una acció. La segona també és una acció, però és més ineficient perquè a cada iteració esborra i inserta elements. En cas contrari, la millor solució és la quarta.