Estructures de dades bàsiques

Seqüències

Xavier Sáez Pous

PID_00146763

Material docent de la UOC

Universitat Oberta de Catalunya
www.uoc.edu

Primera edició: setembre 2010

Av. Tibidabo, 39-43, 08035 Barcelona
Disseny: Manel Andreu
Realització editorial: Eureca Media, SL
ISBN: 978-84-693-4240-4
Dipòsit legal: B-33.174-2010

Índex

Introducció

Una **seqüència** és un conjunt d'elements disposats en un ordre específic. Fruit d'aquesta ordenació, donat un element de la seqüència parlarem del *predecessor* (com l'element anterior) i del *successor* (com l'element següent).

Donada la seqüència $<v_0\ v_1\ ...\ v_i\ v_{i+1}\ ...\ v_n>$, es diu que l'element v_{i+1} és el **successor** de l'element v_i i que l'element v_i és el **predecessor** de l'element v_{i+1}. De la mateixa manera, direm que v_0 és el **primer** element de la seqüència i v_n és l'**últim**.

La majoria d'algorismes que desenvoluparem se centraran en la repetició d'un conjunt d'accions sobre una seqüència de dades. D'això la importància de saber treballar amb les seqüències. Sense anar més lluny, en assignatures anteriors ja heu estudiat els esquemes de recorregut i cerca en una seqüència.

Però, a més, a part de la importància del tractament de les seqüències, en aquest mòdul esbrinarem com s'emmagatzemen de manera que posteriorment hi puguem accedir seqüencialment.

Un **tipus de dades** consta d'un conjunt de valors i d'una sèrie d'operacions que s'hi poden aplicar. Les operacions han de complir certes propietats que en determinaran el comportament.

Les operacions necessàries per a treballar amb una seqüència són:

- Crear la seqüència buida.
- Inserir un element dins de la seqüència.
- Esborrar un element de la seqüència.
- Consultar un element de la seqüència.

El comportament que s'estableixi per a cadascuna de les operacions (on s'insereix un element nou? quin element s'esborra? quin element es pot consultar?) definirà quin **tipus de dades** necessitarem.

Un **tipus abstracte de dades** (abreujadament **TAD**) és un **tipus de dades** al qual s'ha afegit el concepte **abstracció** per a indicar que la implementació del tipus és invisible per als usuaris del tipus.

Un TAD qualsevol constarà de dues parts:

- **Especificació.** En què es definirà completament i sense ambigüitats el comportament de les operacions del tipus.

- **Implementació.** En què es decidirà una representació pels valors del tipus i es codificaran les operacions a partir d'aquesta representació. Donat un tipus, podem trobar diverses implementacions, cadascuna de les quals haurà de seguir l'especificació del tipus.

Així, doncs, l'únic coneixement necessari per a utilitzar un TAD és l'*especificació*, ja que explica les propietats o el comportament de les operacions del tipus.

Les operacions del TAD es dividiran en **constructores** (retornen un valor del mateix tipus) i **consultores** (retornen un valor d'un altre tipus). Dins de les operacions constructores, s'anomenaran **generadores** les que formin part del conjunt mínim d'operacions necessàries per a generar qualsevol valor del tipus, mentre que la resta s'anomenaran **modificadores**.

En aquest mòdul, ens centrarem en els tres TAD més típics per representar les seqüències:

- Piles
- Cues
- Llistes

L'explicació de cada tipus seguirà la mateixa estructura. Primer, es presentarà el tipus d'una manera intuïtiva per a entendre el concepte. A continuació, es descriurà el comportament de les operacions del tipus d'una manera formal. I, per acabar, es desenvoluparà una implementació del tipus acompanyada d'algun exemple per a veure'n l'ús.

Objectius

A la fi d'aquest mòdul, haureu assolit els objectius següents:

1. Conèixer els TAD *pila*, *cua* i *llista*; i saber quines propietats els diferencien.
2. Donat un problema, decidir quin és el TAD més adequat per a emmagatzemar les dades i saber-lo utilitzar.
3. Ser capaç d'implementar qualsevol dels TAD presentats en el mòdul, utilitzant vectors o punters. També ser capaç d'implementar noves operacions en aquests TAD.
4. Entendre els costos (temporals i espacials) que es deriven de les diferents implementacions d'un TAD, i valorar segons aquests criteris quina és la millor implementació en cada situació.
5. Codificar algorismes fent servir els TAD presentats.

1. Piles

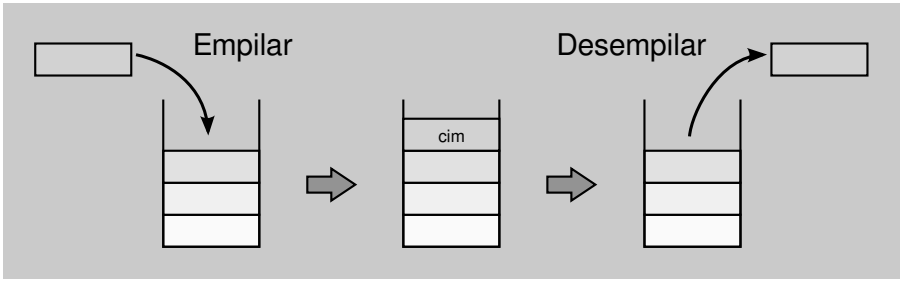
1.1. Representació

Una **pila** és un TAD que es caracteritza perquè l'últim element a entrar és el primer a sortir.

En anglès, una pila s'acostuma a anomenar amb les sigles LIFO (*last in first out*).

La definició de pila ens indica que totes les operacions treballen sobre el mateix extrem de la seqüència. En altres paraules, els elements es treuen de l'estructura en l'ordre invers a l'ordre en què s'han inserit, ja que l'únic element de la seqüència que es pot obtenir és el darrer (figura 1).

Figura 1. Representació d'una pila



Exemples de piles

En la nostra vida diària podem veure aquest comportament molt sovint. Per exemple, si apilem els plats d'una vaixella, únicament podrem agafar el darrer plat afegit a la pila de plats, perquè qualsevol intent d'agafar un plat del mig de la pila (com el plat fosc) acabarà en una trencadissa. Un altre exemple el tenim en els jocs de cartes, en què generalment agafem les cartes (una a una) del cim de la baralla (figura 2).

Figura 2. Pila de plats i de cartes

En el món informàtic també trobem piles, com per exemple, en els navegadors web. Cada cop que accedim a una nova pàgina, el navegador l'afegeix a una pila de pàgines visitades, de manera que quan seleccionem l'opció `Anterior` el navegador agafa la pàgina que es troba al cim de la pila perquè justament aquesta és la darrera pàgina visitada.

Un altre exemple el tenim en els processadors de textos, en què els canvis introduïts en el text també s'emmagatzemen en una pila. Cada cop que premem la combinació de tecles `Ctrl+Z` desfem l'últim canvi introduït, mentre que cada cop que premem la combinació `Ctrl+Y` tornem a afegir a la pila el darrer canvi desfet.

1.2. Operacions

A la taula 1 teniu les operacions bàsiques per a treballar amb piles.

Taula 1

Nom	Descripció
crear	Crea una pila buida
empilar	Insereix un element a la pila
desempilar	Treu l'element situat al cim de la pila
cim	Retorna l'element situat al cim de la pila
buida	Retorna *cert* si la pila està buida i *fals* altrament

Les operacions del TAD pila es classifiquen en:

- **Operacions constructores:** `crear`, `empilar` i `desempilar`.
- **Operacions consultores:** `cim` i `buida`.

Recordeu

Recordeu que les operacions constructures són les operacions que modifiquen l'estat de la pila. Mentre que les operacions consultores són les operacions que consulten l'estat de la pila sense modificar-la.

L'**estat d'una pila** està definit pels elements que conté la pila i l'ordre en què estan emmagatzemats. Tot estat és el resultat d'una seqüència de crides a operacions constructores.

Així mateix, les operacions constructores es classifiquen en:

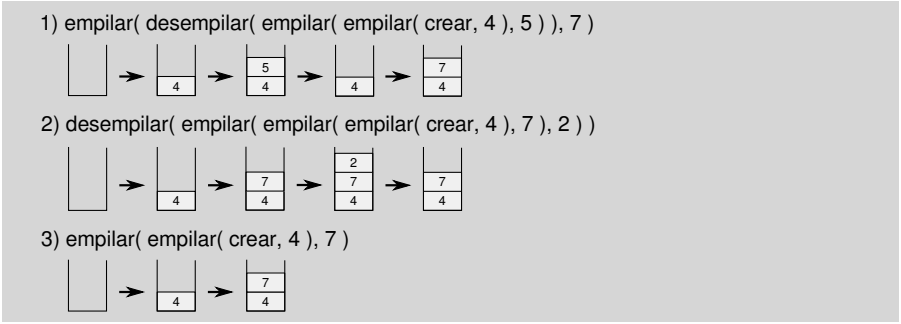
- **Operacions generadores:** `crear` i `empilar`, perquè són imprescindibles per a aconseguir qualsevol estat de la pila.

- **Operacions modificadores:** `desempilar`, perquè modifica l'estat de la pila traient l'element del cim, però no és una operació imprescindible per a construir una pila.

Donat un estat de la pila, s'hi pot arribar a través de diverses seqüències de crides a operacions constructores, però d'entre totes les seqüències **només n'hi haurà una que estigui formada exclusivament per operacions generadores** (figura 3).

Observeu que la tercera seqüència de crides de la figura 3 és la mínima per a arribar a l'estat desitjat, ja que no podem eliminar cap de les crides que la formen. Per tant, tal com s'ha mencionant anteriorment, empilar i crear són les operacions generadores.

Figura 3. Exemples de seqüències d'operacions que produeixen una pila amb l'estat <4,7>.



Un cop establertes les operacions del tipus cal especificar-ne el comportament d'una manera formal mitjançant la **signatura**.

A continuació, introduïm la **signatura** del tipus pila que estableix per a cada operació quins són els paràmetres, el resultat, i també les equacions que en reflecteixen el comportament i les condicions sota les quals l'operació pot provocar errors.

tipus pila (elem)
  operacions
    crear: → pila
    empilar: pila elem → pila
    desempilar: pila → pila
    cim: pila → elem
    buida: pila → booleà
  errors
    desempilar(crear)
    cim(crear)
  equacions $\forall p \in pila;\ \forall e \in elem$
    desempilar(empilar(p,e)) = p
    cim(empilar(p,e)) = e
    buida(crear) = cert
    buida(empilar(p,e)) = fals
ftipus

En la signatura del tipus,

- Parlem de **pila(elem)** per denotar que la pila emmagatzema elements del tipus genèric *elem*.

- Les seccions **errors** i **equacions** defineixen el comportament de les operacions cobrint tots els casos possibles: davant una pila buida (representada

per `crear`) o davant una pila no buida (representada per `empilar(p,e)`, ja que la pila resultant conté almenys un element).

- La secció **equacions** explica el comportament normal de les operacions com, per exemple: el resultat de desempilar un element d'una pila és la pila que hi havia abans d'empilar el darrer element, el cim d'una pila és l'últim element empilat, i una pila només està buida si no hi ha cap element empilat.

- La secció **errors** determina quines situacions produeixen errors en les operacions. En el cas de les piles, treure o consultar el cim d'una pila buida produeix un error.

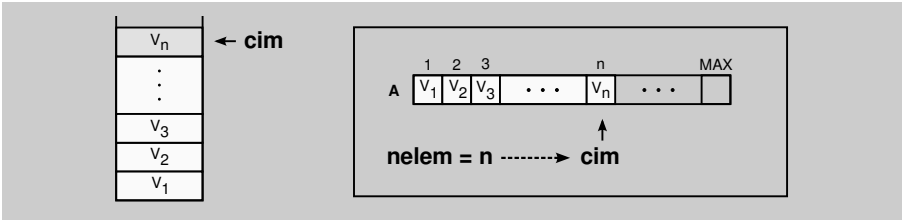
Les **funcions parcials** són les funcions que no estan definides per tot el domini dels paràmetres d'entrada. En el cas de les piles, les operacions desempilar i cim són funcions parcials, perquè hi ha situacions en què produeixen errors i, per tant, no hi són definides.

1.3. Implementació

La implementació més senzilla d'una pila és utilitzant un **vector** d'elements. Com que els vectors no tenen una dimensió infinita, caldrà definir el nombre màxim d'elements (**MAX**) que s'hi podrà emmagatzemar i també caldrà afegir una operació (`plena`) per a comprovar si la pila és plena abans d'empilar un nou element. D'aquest manera, evitarem l'error d'empilar un element quan no quedi espai lliure en el vector.

La figura 4 mostra la representació d'una pila mitjançant un vector. El primer element de la pila s'emmagatzema en la primera posició del vector, el segon element en la segona posició... i així fins arribar al darrer element.

Figura 4. Implementació d'una pila amb un vector



La implementació inclourà, a part del vector en què es guarden els elements (A), un atribut enter (`nelem`) que ens indicarà el nombre d'elements que hi ha a la pila en tot moment i, implícitament, la posició del vector en què es troba el cim de la pila. Aquest apuntador també dividirà el vector en dues parts: part ocupada i part lliure.

```
tipus
  pila = tupla
           A : taula [MAX] de elem;
           nelem : enter;
  ftupla
ftipus
```

Donada la representació del tipus, el pas següent és implementar les operacions del tipus pila:

```
funcio crear() : pila
  var p : pila fvar
  p.nelem := 0;
  retorna p;
ffuncio

funcio empilar(p : pila; e : elem) : pila
  si p.nelem = MAX llavors
      error {pila plena};
  sino
      p.nelem := p.nelem + 1;
      p.A[p.nelem] := e;
  fsi
  retorna p;
ffuncio

funcio desempilar(p : pila) : pila
  si p.nelem = 0 llavors
      error {pila buida};
  sino
      p.nelem := p.nelem – 1;
  fsi
  retorna p;
ffuncio

funcio cim(p : pila) : elem
  var e : elem fvar
  si p.nelem = 0 llavors
      error {pila buida};
  sino
      e := p.A[p.nelem];
  fsi
  retorna e;
ffuncio
```

funcio buida(*p* : **pila**) : **boolea**
  retorna *p.nelem* = 0;
ffuncio

funcio plena(*p* : **pila**) : **boolea**
  retorna *p.nelem* = *MAX*;
ffuncio

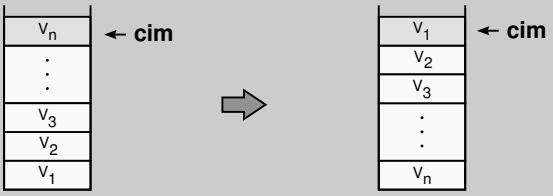
Resta fer una anàlisi aproximada de l'eficiència temporal i espacial de la implementació per a valorar-ne la validesa. Per a fer-ho es calcularà el cost asimptòtic Θ de cada operació.

El **cost temporal** de les operacions és òptim, Θ(1), ja que no hi ha cap bucle en el cos de les funcions. D'altra banda, el **cost espacial** és força pobre, Θ(*MAX*), ja que el vector reserva un espai (concretament, *MAX* posicions) independentment del nombre d'elements que emmagatzemi la pila en un instant donat.

1.4. Exemple

Es demana definir una nova operació que estengui el TAD pila. L'operació s'anomenarà ***invertir*** *i capgirarà el contingut de la pila. És a dir, el primer element de la pila passarà a ser l'últim, el segon element passarà a ser el penúltim... i així successivament (figura 5).*

Figura 5. Operació `invertir`



Per implementar la nova operació, aprofitarem el comportament **LIFO** de les piles* per llegir els elements de la pila en l'ordre invers d'arribada. Així, doncs, simplement s'haurà de desempilar cada element de la pila origen i a continuació empilar-lo a la pila resultat. D'aquesta manera, obtindrem fàcilment una pila on els elements estaran en ordre invers a com estaven en la pila original.

***El darrer element que entra és el primer que en surt.**

L'algorisme de l'operació **invertir** constarà dels passos següents:

1) Crear la pila resultat on s'empilaran els elements.
2) Llegir el cim de la pila origen i empilar-lo a la pila resultat.
3) Desempilar l'element acabat de llegir del cim de la pila origen.
4) Tornar al punt 2 mentre la pila origen no estigui buida.

Finalment, implementem l'operació:

```
funcio invertir(p : pila) : pila
   var pinv : pila fvar
   pinv := crear();
   mentre ¬buida(p) fer
            pinv := empilar(pinv,cim(p));
            p := desempilar(p);
   fmentre
   retorna pinv;
ffuncio
```

2. Cues

2.1. Representació

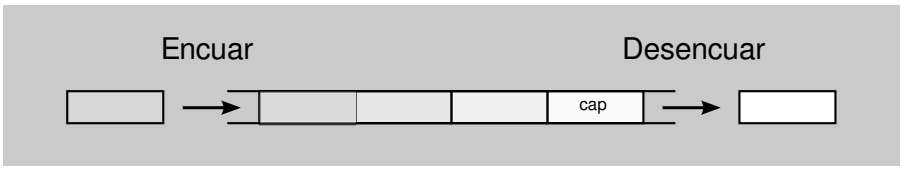
Les cues són el segon TAD que estudiarem per representar les seqüències. Bàsicament, les cues es diferencien de les piles en l'extracció de les dades.

Una **cua** és un TAD caracteritzat pel fet que el primer element a entrar és el primer a sortir.

En anglès, una cua s'acostuma a anomenar amb les sigles FIFO (*first in first out*).

La definició de cua ens indica que les operacions treballen sobre els dos extrems de la seqüència: un extrem per a afegir els elements i l'altre extrem per a consultar-los i/o treure'ls. En altres paraules, els elements s'extreuen en el mateix ordre en què s'han inserit prèviament, ja que els elements s'insereixen pel final de la seqüència i s'extreuen per la capçalera (figura 6).

Figura 6. Representació d'una cua



Exemples de cues

Les cues apareixen sovint en la nostra vida diària... Sense anar més lluny, podem afirmar que passem una part de la nostra vida fent cues: per a comprar l'entrada en un cinema, per a pagar a la caixa d'un supermercat, per a visitar-nos a cal metge... La idea sempre és la mateixa: s'atén la primera persona de la cua, que és la que fa més estona que s'espera, i un cop atesa llavors surt de la cua i la persona següent passa a ser la primera de la cua (figura 7).

Figura 7. Cua de motxilles per a entrar a un alberg

Si en el món real és habitual veure cues, en el món informàtic encara ho és més. Quan el sistema operatiu ha de **gestionar l'accés a un recurs compartit** (processos que volen executar-se en la CPU, treballs que s'envien a una impressora, descàrrega de fitxers, etc.), una de les estratègies més utilitzades és organitzar les peticions per mitjà de cues. Per exemple, la figura 8 ens mostra una captura d'una cua d'impressió en un instant donat. En aquest cas, la tasca 321 s'està imprimint perquè és la primera en la cua, mentre que la tasca 326 serà l'última a imprimir-se perquè ha estat la darrera a arribar.

Figura 8. Captura d'una cua d'impressió

2.2. Operacions

Donada la representació d'una cua, a la taula 2 definim les operacions per a treballar-hi.

Taula 2

Nom	Descripció
crear	Crea una cua buida
encuar	Insereix un element a la cua
desencuar	Treu l'element situat al principi de la cua
cap	Retorna l'element situat al principi de la cua
buida	Retorna *cert* si la cua està buida i *fals* altrament

Com en el TAD anterior, les operacions del TAD cua es classifiquen segons el seu comportament en generadores, modificadores i consultores:

- **Operacions constructores:**
  - **Operacions generadores:** `crear` i `encuar`.
  - **Operacions modificadores:** `desencuar`.
- **Operacions consultores:** `cap` i `buida`.

La **signatura del TAD *cua*** especifica d'una manera formal el comportament de les operacions per tots els casos possibles:

tipus cua (elem)
  operacions
    crear: → cua
    encuar: cua elem → cua
    desencuar: cua → cua
    cap: cua → elem
    buida: cua → boolea

errors
desencuar(crear)
cap(crear)
equacions $\forall c \in cua$; $\forall e \in elem$
desencuar(encuar(crear,e)) = crear
[¬buida(c)] ⇒ desencuar(encuar(c,e)) = encuar(desencuar(c),e)
cap(encuar(crear,e)) = e
[¬buida(c)] ⇒ cap(encuar(c,e)) = cap(c)
buida(crear) = cert
buida(encuar(c,e)) = fals
ftipus

El comportament de les operacions `desencuar` i `cap` es resumeix en tres casos:

1) sobre una cua buida (`crear`),

2) sobre una cua amb un únic element (`encuar(crear,e)`) i

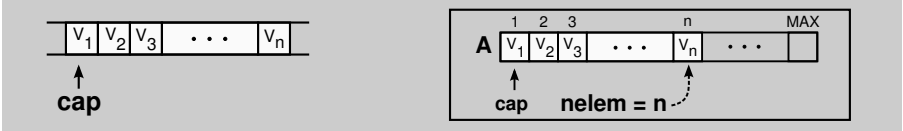
3) sobre una cua amb més d'un element (`encuar(c,e)`, on `c` no és una cua buida).

2.3. Implementació

Un altre cop, l'opció més senzilla per a implementar una cua és utilitzar un **vector**. Per tant, com que tornem a tenir una implementació fitada, cal definir el nombre màxim d'elements que pot emmagatzemar (`MAX`) i una operació que ens indiqui quan és `plena`.

Al vector d'elements, hi afegim un atribut enter (`nelem`) per conèixer el nombre d'elements que hi ha en la cua en tot moment. Tal com es veu en la figura 9, la primera posició del vector serà sempre l'inici de la cua, mentre que l'atribut `nelem` ens indicarà quin és l'últim element i, implícitament, l'inici de l'espai lliure.

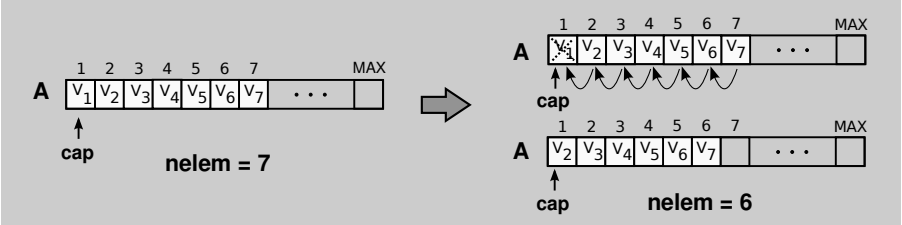
Figura 9. Implementació d'una cua amb un vector



Aquesta representació presenta un **greu problema d'ineficiència** en l'operació **desencuar**. Cada cop que s'elimina el primer element de la cua, el manteniment de la condició segons la qual "*l'inici de la cua és sempre la primera*

posició del vector" ens obliga a desplaçar tots els elements una posició (com es pot veure en la figura 10). Aquest desplaçament comporta que l'operació tingui un cost lineal $\Theta(nelem)$.

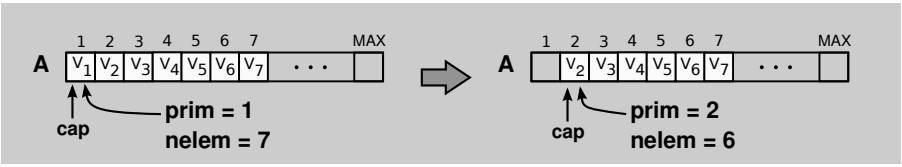
Figura 10. Operació **desencuar**



Per millorar la implementació anterior, afegim un nou atribut enter (`prim`) que apuntarà al primer element de la cua. De manera que els dos extrems de la cua es desplaçaran sobre el vector, mentre que els elements emmagatzemats no es mouran (figura 11).

D'aquesta manera, la nova representació soluciona el problema de la ineficiència en l'operació `desencuar`, perquè, sempre que se suprimeixi un element de la cua, només s'haurà d'incrementar l'atribut `prim` per a apuntar a l'element següent.

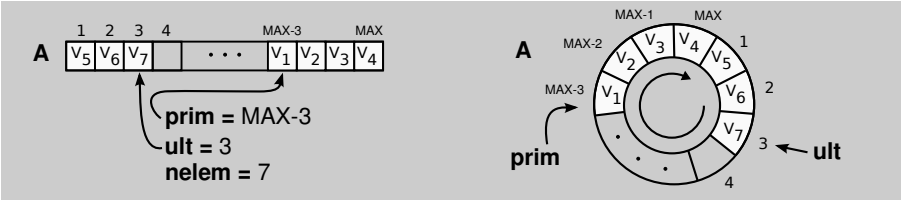
Figura 11. Operació **desencuar** amb el camp `prim`



Encara més, aquesta solució aconsegueix que totes les operacions tinguin un cost òptim $\Theta(1)$, però presenta un altre problema greu: **desaprofita molt d'espai**, ja que no es reutilitzen les posicions alliberades en la part inicial del vector. Fins al punt, que si encuem i desencuem molts cops (fruit d'un ús normal de la cua) ens podem quedar sense espai encara que la cua només tingui un element.

Per reaprofitar aquestes posicions inicials sense moure els elements, considerarem que el vector és una **estructura circular**, és a dir, després de la darrera posició del vector vindrà la primera posició. Ens trobem amb una estructura sense principi ni final, tal com es veu en la figura 12.

Figura 12. Representació d'una cua en un vector circular



Per **gestionar el vector circularment**, recorrem a l'operació **mòdul** que calcula el residu de la divisió de dos nombres enters. Donada la posició *p* d'un vector amb mida *m*, el càlcul per a esbrinar la posició següent del vector és:

$$(p \bmod m) + 1$$

Per tant, la representació escollida serà la del **vector circular amb el punter** `prim`. El **cost temporal** de les operacions serà òptim $\Theta(1)$, mentre que el **cost espacial** serà igual de pobre que amb les piles $\Theta(MAX)$ (ja que el vector es crea inicialment amb la mida *MAX*). Malgrat això, la implementació circular aconsegueix omplir tot el vector gràcies a la reutilització de les posicions alliberades per a emmagatzemar-hi nous elements.

```
tipus
  cua = tupla
          A : taula [MAX] de elem;
          prim,ult,nelem : enter;
  ftupla
ftipus
```

Donada la representació del tipus, mostrem la implementació de les operacions del tipus cua:

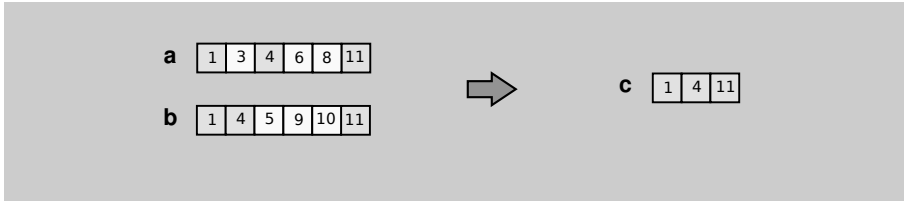
```
funcio crear() : cua
  var c : cua fvar
  c.prim := 1;
  c.ult := 0;
  c.nelem := 0;
  retorna c;
ffuncio

funcio encuar(c : cua; e : elem) : cua
  si c.nelem = MAX llavors
      error {cua plena};
  sino
      c.ult := (c.ult mod MAX) + 1;
      c.A[c.ult] := e;
      c.nelem := c.nelem + 1;
  fsi
  retorna c;
ffuncio
```

```
funcio desencuar(c : cua) : cua
  si c.nelem = 0 llavors
      error {cua buida};
  sino
      c.prim := (c.prim mod MAX) + 1;
      c.nelem := c.nelem – 1;
  fsi
  retorna c;
ffuncio

funcio cap(c : cua) : elem
  var e : elem fvar
  si c.nelem = 0 llavors
      error {cua buida};
  sino
      e := c.A[c.prim];
  fsi
  retorna e;
ffuncio

funcio buida(c : cua) : boolea
  retorna c.nelem = 0;
ffuncio

funcio plena(c : cua) : boolea
  retorna c.nelem = MAX;
ffuncio
```

2.4. Exemple

Es demana definir una nova operació, anomenada ***interseccio****, que estengui el TAD cua. Aquesta operació rebrà com a entrada dues cues amb els elements ordenats de manera creixent i retornarà una cua ordenada estrictament creixent només amb els elements que apareguin en les dues cues d'entrada (figura 13).*

Figura 13. Exemple de l'operació `interseccio`



El fet que la cua estigui ordenada ens evita haver de recórrer tota la cua cada cop que volem comprovar si conté un element donat. En altres paraules, si el primer element de la cua *a* és més petit que el primer element de la cua *b*, segur

que no trobarem un element igual en tota la cua *b*, perquè tots els elements de la cua *b* seran més grans que el primer element de la cua *a*.

Per tant, amb aquesta informació establim que l'estratègia de l'operació **interseccio** serà la següent:

1) Crear la cua en què s'encuaran els elements comuns a les dues cues d'entrada.

2) Mentre les dues cues no estiguin buides, llegir el primer element de les dues cues.

3) Si els dos elements són iguals, llavors encuar l'element a la cua resultat i desencuar un element de les dues cues. Altrament, si els dos elements no són iguals, llavors només desencuar l'element més petit.

4) Tornar al punt 2 mentre no s'hagi buidat alguna de les dues cues.

Finalment, implementem l'operació seguint l'esquema anterior:

```
funcio interseccio(c1 : cua; c2 : cua) : cua
  var c3 : cua fvar
  c3 := crear();
  mentre ¬buida(c1) ∧ ¬buida(c2) fer
          si cap(c1) = cap(c2) llavors
              c3 := encuar(c3,cap(c1));
              c1 := desencuar(c1);
              c2 := desencuar(c2);
          sino si cap(c1) < cap(c2) llavors
                  c1 := desencuar(c1);
              sino
                  c2 := desencuar(c2);
              fsi
          fsi
  fmentre
  retorna c3;
ffuncio
```

3. Llistes

3.1. Representació

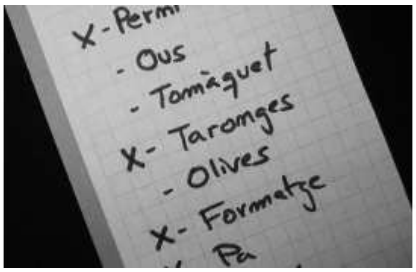
Les llistes són el darrer TAD que estudiarem per representar les seqüències. Mentre que en els TAD anteriors hem vist com les operacions treballaven només amb els extrems de la seqüència, les llistes ens oferiran la possibilitat d'accedir a qualsevol punt d'aquesta.

Una **llista** és un TAD caracteritzat pel fet que permet afegir, esborrar o consultar qualsevol element de la seqüència. És l'estructura lineal més flexible, fins al punt de considerar els TAD **pila** i **cua** casos particulars del TAD **llista**.

Exemples de llistes

També en aquest cas, trobem llistes en la nostra vida diària. Per exemple, la llista de la compra. Quan som al supermercat, generalment eliminem els articles a mesura que els trobem en el recorregut que seguim amb el carro, i que no ha de coincidir necessàriament amb l'ordre en què els hem escrit en la llista (figura 14).

Figura 14. Una llista de la compra



Des del punt de vista informàtic també trobem exemples, com els editors de textos. Quan escrivim un codi, en el fons editem una llista de paraules dins d'una llista de línies. Parlem de llistes, perquè en qualsevol moment ens podem desplaçar sobre qualsevol paraula del fitxer per a modificar-la o inserir-ne de noves.

Hi ha diferents models de llistes, un dels més habituals és l'anomenat **llista amb punt d'interès**. Aquesta llista conté un **element distingit** que serveix de referència per a aplicar les operacions. L'element distingit és apuntat per l'anomenat **punt d'interès**, el qual pot ser desplaçat.

El punt d'interès divideix una seqüència en dos fragments, que alhora també són seqüències. Per tant, donada una llista *l* qualsevol, aquesta es pot dividir en una seqüència situada a l'esquerra del punt d'interès (*s*) i una altra seqüència que va del punt d'interès (element distingit, *e*) cap a la dreta (*et*). Podem representar aquesta unió de dues seqüències per a formar la llista *l* de la manera següent: $l =< s,et >$. La seqüència buida (és a dir, sense cap element) es representarà amb la lletra λ.

3.2. Operacions

A la taula 3 teniu les operacions per treballar amb les llistes.

Taula 3

Nom	Descripció
crear	Crea una llista buida
inserir	Insereix un element a la llista davant del punt d'interès
esborrar	Treu l'element distingit i desplaça el punt d'interès a l'element següent
actual	Retorna l'element distingit
buida	Retorna *cert* si la llista és buida i *fals* altrament
principi	Situa el punt d'interès sobre el primer element de la llista
avançar	Desplaça el punt d'interès a l'element següent
fi	Retorna *cert* si el punt d'interès és a la dreta de tot (final) de la llista i *fals* altrament

Podem veure que a part de les típiques operacions per a treballar amb seqüències, aquest cop s'han afegit operacions per a canviar l'element distingit i així poder desplaçar el punt d'interès: `principi` i `avançar`.

El pas següent és determinar la **signatura del tipus llista**:

tipus llista (elem)
  operacions
    crear: → llista
    inserir: llista elem → llista
    esborrar: llista → llista
    actual: llista → elem
    buida: llista → booleà
    principi: llista → llista
    avançar: llista → llista
    fi: llista → booleà
  errors
    esborrar(<s,λ>)
    avançar(<s,λ>)
    actual(<s,λ>)

equacions $\forall < s,t > \in$ *llista*; $\forall e \in$ *elem*
crear = <λ,λ>
inserir(<s, t>, e) = <se, t>
esborrar(<s, et>) = <s, t>
principi(<s, t>) = <λ, st>
avançar(<s, et>) = <se, t>
actual(<s, et>) = e
[t = λ] $\Rightarrow$ fi(<s, t>) = cert
[t $\neq \lambda$] $\Rightarrow$ fi(<s, t>) = fals
buida(crear) = cert
buida(inserir(<s, t>,e)) = fals
ftipus

En aquest cas, el conjunt d'**operacions constructores generadores** està format per `crear`, `inserir` i `principi`. Per tant, la mínima seqüència de crides que necessitarem per a obtenir la llista `<ho, la>` és una combinació d'aquestes operacions:

```
inserir(inserir(principi(inserir(inserir(crear,l),a)),h),o)
```

3.3. Implementació

La implementació de les llistes amb punt d'interès es pot fer de diverses maneres:

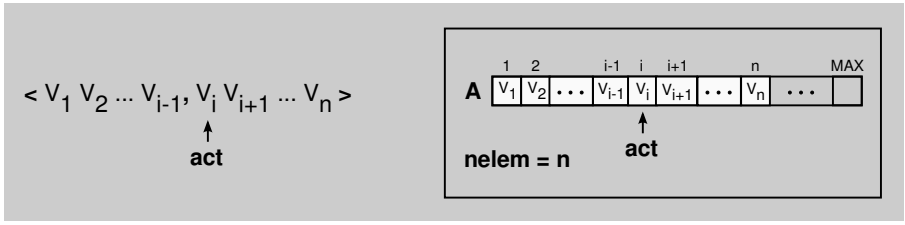
- **Implementació seqüencial.** Els elements de la llista s'emmagatzemen en un vector respectant la norma que els elements consecutius ocupen posicions consecutives.

- **Implementació encadenada.** Els elements s'emmagatzemen sense seguir la norma anterior, ja que cada element del vector guarda la posició on es troba el següent element de la llista.

3.3.1. Implementació seqüencial

La representació requereix tres elements: un vector (`A`), un indicador del nombre d'elements emmagatzemats (`nelem`) i un indicador del punt d'interès o element distingit (`act`).

Figura 15. Implementació seqüencial d'una llista amb un vector



L'indicador `nelem` ens serveix per a controlar que no sobrepassem el nombre màxim d'elements (`MAX`) que pot emmagatzemar el vector. Mentre que l'indicador `act` apunta a l'element distingit de la llista (figura 15).

```
tipus
  llista = tupla
             A : taula [MAX] de elem;
             act,nelem : enter;
  ftupla
ftipus
```

Donada la representació del tipus, implementem les operacions del tipus llista:

```
funcio crear() : llista
  var l : llista fvar
  l.act := 1;
  l.nelem := 0;
  retorna l;
ffuncio

funcio inserir(l : llista; e : elem) : llista
  var i : enter fvar
  si l.nelem = MAX llavors
      error {llista plena};
  sino
      per i := l.nelem fins l.act pas -1 fer
          l.A[i + 1] := l.A[i];
      fper
      l.A[l.act] := e;
      l.nelem := l.nelem + 1;
      l.act := l.act + 1;
  fsi
  retorna l;
ffuncio
```

```
funcio esborrar(l : llista) : llista
  var i : enter fvar
  si l.act > l.nelem llavors
      error {fi de llista o llista buida};
  sino
      per i := l.act fins l.nelem – 1 fer
          l.A[i] := l.A[i + 1];
      fper
      l.nelem := l.nelem – 1;
  fsi
  retorna l;
ffuncio

funcio principi(l : llista) : llista
  l.act := 1;
  retorna l;
ffuncio

funcio avançar(l : llista) : llista
  si l.act > l.nelem llavors
      error {fi de llista o llista buida};
  sino
      l.act := l.act + 1;
  fsi
  retorna l;
ffuncio

funcio actual(l : llista) : elem
  var e : elem fvar
  si l.act > l.nelem llavors
      error {fi de llista o llista buida};
  sino
      e := l.A[l.act];
  fsi
  retorna e;
ffuncio

funcio fi(l : llista) : boolea
  retorna l.act > l.nelem;
ffuncio

funcio buida(l : llista) : boolea
  retorna l.nelem = 0;
ffuncio
```

funcio plena(*l* : **llista**) : **boolea**
  retorna *l.nelem* = *MAX*;
ffuncio

Però aquesta implementació del TAD *llista* presenta **diversos problemes** quan revisem les operacions `inserir` i `esborrar`:

- Les operacions presenten desplaçaments d'elements dins del vector, concretament en les posicions de la dreta del punt d'interès (en els bucles). Aquests desplaçaments fan que el **cost temporal** de les dues operacions sigui lineal, per tant la seva eficiència és molt dolenta tenint en compte la freqüència d'ús (figura 16).

- Si els elements de la llista són **grans**, el seu desplaçament dins del vector (per a no desaprofitar espai) és molt costos.

- La **integració de la llista en una estructura de dades** és complexa. Els elements de la llista són apuntats des de diferents components de l'estructura, de manera que cada desplaçament d'un element del vector implica automàticament actualitzar tots els apuntadors externs al vector que apuntin a l'element desplaçat (figura 17). Com us podeu imaginar, aquesta tasca no és trivial.

Figura 16. Operació `esborrar` en la implementació seqüencial d'una llista

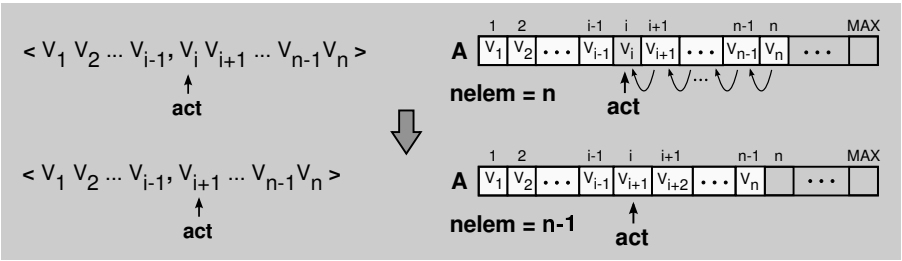
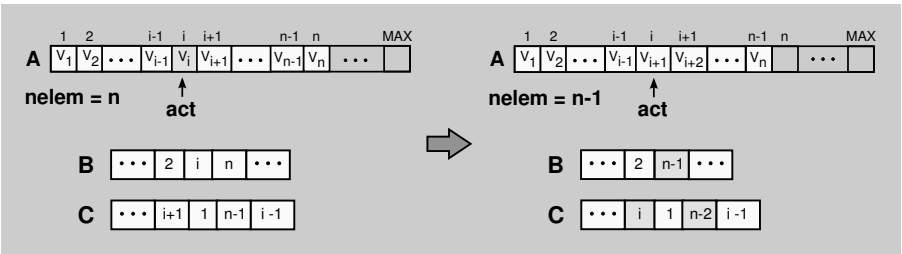


Figura 17. `Esborrar` un element de la llista A afecta les estructures (B i C) que l'apunten.

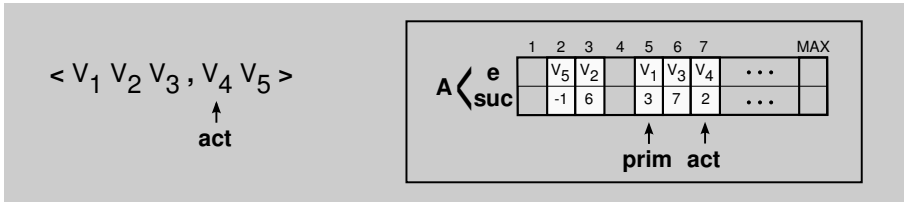


3.3.2. Implementació encadenada

Per trencar el concepte de **seqüencial**, en què el successor d'un element és el que ocupa la posició següent del vector, introduïm a continuació el concepte d'**encadenament**, en què cada element guarda la posició en què es troba el seu successor.

La representació, com es pot veure en la figura 18, emmagatzema per cada posició del vector: l'element (**e**) i un indicador a la posició en què es troba l'element següent (**suc**). Aquest indicador s'anomenarà d'ara endavant **encadenament**.

Figura 18. Implementació encadenada d'una llista amb un vector



La implementació encadenada resol els problemes que patia la representació seqüencial en les operacions `inserir` i `esborrar`, perquè evita desplaçar elements en el vector en no haver-los de mantenir seqüencialment.

Malgrat que el preu a pagar per estalviar aquests desplaçaments és l'emmagatzemament d'un camp extra per cada element, aquest cost sembla assumible especialment en llistes molt volàtils (moltes operacions d'inserir i esborrar elements) o amb elements de grans dimensions (que facin negligible l'espai ocupat per aquest camp extra).

Per tant, en aquest cas, la representació de la llista constarà d'un indicador al primer element (`prim`), un altre a l'element distingit o actual (`act`) i un vector (`A`) amb un indicador per cada element (`e`) de quin és el següent (`suc`). En el cas del darrer element, l'indicador tindrà un **valor nul**, que per conveni sol ser -1 (ja que no existeix una posició negativa en un vector).

```
tipus
  llista = tupla
             A : taula [MAX] de  tupla
                                   e : elem;
                                   suc : enter;
                                 ftupla;
             act,prim : enter;
  ftupla
ftipus
```

A partir d'aquesta representació del tipus, implementem les operacions que ens permetran recórrer la llista i consultar/esborrar/inserir elements en ella.

Però abans, cal pensar com es poden **reaprofitar les posicions del vector que s'han esborrat** per a emmagatzemar nous elements. A continuació, analit-

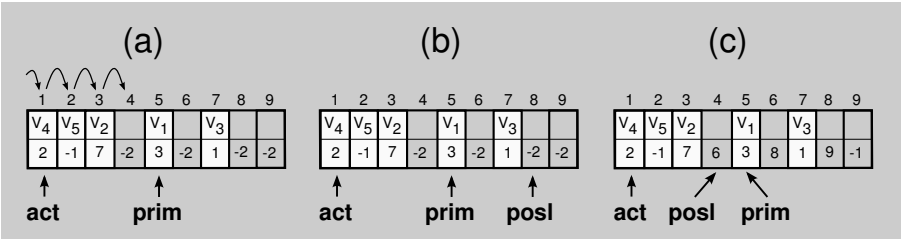
zem algunes de les estratègies que es poden aplicar per a reaprofitar aquestes posicions:

- Marcar totes les posicions del vector com a ocupades o lliures aprofitant el camp `suc`, sempre que no es disposi de cap estructura extra que ens indiqui quines posicions estan lliures. Per diferenciar les posicions lliures triarem un valor no utilitzat en el camp `suc` com, per exemple, -2, ja que les posicions del vector són enters positius i el valor nul és -1. *Problema*: per cada inserció cal fer una cerca d'una posició lliure en el vector i això pot ser molt costos (figura 19a).

- Afegir un indicador (`posl`) que apunti a la primera posició lliure del vector (figura 19b). Quan s'acabin les posicions lliures, s'haurà de reorganitzar el vector desplaçant tots els elements cap a l'esquerra per a deixar a la dreta tots els espais lliures alliberats en l'esborrat d'elements. *Problema*: tal com s'ha comentat en la implementació seqüencial, la reorganització del vector és massa costosa.

- Utilitzar una pila per a gestionar les posicions lliures: **pila de llocs lliures**. Aquesta pila s'implementa sobre el mateix vector de la llista, aprofitant el camp `suc` per a encadenar les posicions lliures i evitar utilitzar més espai (figura 19c). El funcionament és simple: quan s'insereix un element en la llista, s'extreu el primer lloc lliure de la pila (`posl`); mentre que quan s'esborra un element de la llista, la posició alliberada s'afegix a la pila.

Figura 19. Estratègies per a reaprofitar les posicions alliberades del vector



Només queda resoldre un altre detall per a afrontar la implementació de les operacions `inserir` i `esborrar`: **com es pot actualitzar l'indicador** `suc` **de l'element anterior** perquè apunti al nou successor.

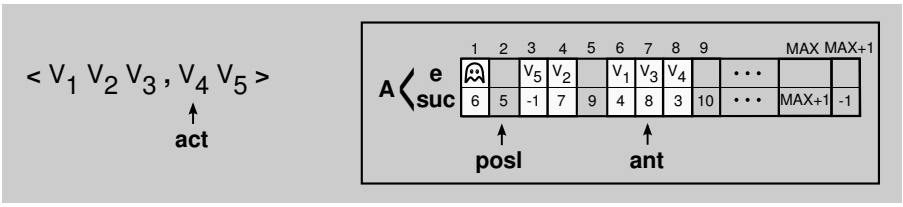
Atès que la representació encadenada no assegura que les dades estiguin seqüencialment en el vector, no és trivial l'accés a l'element anterior de l'actual (o triat) i cal buscar una solució:

- Si no s'afegeix res més a la representació, llavors s'ha de recórrer la llista des del principi buscant l'element que tingui en el camp `suc` la posició de l'element actual. Però això implica que cada crida a les operacions `inserir` i `esborrar` tindrà un cost temporal lineal, $\Theta(n)$.

- Si canviem l'indicador de l'element actual (`act`) per un indicador a l'element anterior de l'actual (`ant`), llavors es podrà accedir a l'element anterior i a l'element actual (successor de l'element anterior) d'una manera trivial i amb un cost temporal constant, $\Theta(1)$.

  Problema: quin és l'element anterior al primer? Per a resoldre-ho tenim dues opcions: o bé emmagatzemar un valor especial (-2) en el camp `ant` quan l'element actual sigui el primer (opció que complica la codificació de les operacions afegint nous casos), o bé guardar sempre en la primera posició del vector un element fictici, anomenat **fantasma** o **sentinella** (figura 20), que no s'esborrarà mai i que farà que l'element actual sempre en tingui un d'anterior (encara que perdem una posició del vector per emmagatzemar dades).

Figura 20. Implementació encadenada d'una llista amb element fantasma



- Si utilitzem una **llista doblement encadenada**, llavors cada posició del vector contindrà l'element, un encadenament a l'element successor i un altre encadenament al predecessor. D'aquesta manera, l'accés a l'element anterior per a actualitzar els indicadors també és trivial. Aquesta opció la veurem més endavant en el subapartat 4.6.2.

Finalment, la representació escollida és la d'una **llista encadenada amb element fantasma i utilitzant un indicador a l'element anterior de l'actual** (`ant`), perquè ens permet codificar totes les operacions de la signatura (menys `crear`) amb un cost constant, $\Theta(1)$.

Fixeu-vos que la llista podrà emmagatzemar fins a `MAX` elements, ja que s'ha sumat una posició `MAX` + 1 per a emmagatzemar el fantasma:

```
tipus
  llista = tupla
             A : taula [MAX + 1] de  tupla
                                       e : elem;
                                       suc : enter;
                                     ftupla;
             ant,posl : enter;
  ftupla
ftipus
```

Tal com s'ha mencionant abans, aquesta representació ens permet implementar totes les operacions amb un **cost temporal** constant, excepte l'operació `crear`, que tindrà un cost lineal en haver de crear la pila d'espais lliures i haver-la d'omplir amb totes les posicions del vector:

```
funcio crear() : llista
  var l : llista fvar
  l.ant := 1;
  l.A[l.ant].suc := –1;     {insereix element fantasma}
  per i := 2 fins MAX fer
      l.A[i].suc := i + 1;
  fper
  l.A[MAX + 1].suc := –1;
  l.posl := 2;
  retorna l;
ffuncio

funcio inserir(l : llista; e : elem) : llista
  var tmp : enter fvar
  si l.posl = –1 llavors
      error {llista plena};
  sino
       tmp := l.posl;
       l.posl := l.A[l.posl].suc;
       l.A[tmp].e := e;
       l.A[tmp].suc := l.A[l.ant].suc;
       l.A[l.ant].suc := tmp;
       l.ant := tmp;
  fsi
  retorna l;
ffuncio

funcio esborrar(l : llista) : llista
  var tmp : enter fvar
  si l.A[l.ant].suc = –1 llavors
      error {fi de llista o llista buida};
  sino
       tmp := l.A[l.ant].suc;
       l.A[l.ant].suc := l.A[tmp].suc;
       l.A[tmp].suc := l.posl;
       l.posl := tmp;
  fsi
  retorna l;
ffuncio
```

```
funcio principi(l : llista) : llista
  l.ant := 1;
  retorna l;
ffuncio

funcio avançar(l : llista) : llista
  si l.A[l.ant].suc = –1 llavors
      error {fi de llista o llista buida};
  sino l.ant := l.A[l.ant].suc;
  fsi
  retorna l;
ffuncio

funcio actual(l : llista) : elem
  var e : elem fvar
  si l.A[l.ant].suc = –1 llavors
      error {fi de llista o llista buida};
  sino
       e := l.A[l.A[l.ant].suc].e;
  fsi
  retorna e;
ffuncio

funcio fi(l : llista) : boolea
  retorna l.A[l.ant].suc = –1;
ffuncio

funcio buida(l : llista) : boolea
  retorna l.A[1].suc = –1;
ffuncio

funcio plena(l : llista) : boolea
  retorna l.posl = –1;
ffuncio
```

3.4. Exemple

Quan parlem dels esquemes de programació sobre seqüències, tots coneixem els esquemes de recorregut i cerca. Tot seguit, implementem aquests dos esquemes sobre una llista "l".

Per començar, recordem que:

- l'esquema de recorregut consisteix a aplicar un `tractament` a tots els elements de la llista,

- l'esquema de cerca consisteix a situar el punt d'interès sobre el primer element de la llista que compleix una `propietat`.

Així, doncs, ja podem implementar l'esquema de recorregut:

```
l := principi(l);
mentre ¬fi(l) fer
        tractament(actual(l));
        l := avançar(l);
fmentre
```

Mentre que la implementació de l'esquema de cerca serà:

```
l := principi(l);
ok := FALS;
mentre ¬fi(l) ∧ ¬ok fer
        si propietat(actual(l)) llavors
            ok := CERT;
        sino
            l := avançar(l);
        fsi
fmentre
```

4. Punters

4.1. Problemes dels vectors

Totes les implementacions fetes fins ara en el mòdul s'han basat en els vectors perquè són fàcils d'utilitzar, però també presenten alguns problemes que cal tenir en compte:

- Els vectors es creen amb una mida que restarà fixa durant tota l'execució, per tant, **no s'adapten a l'evolució dinàmica del programa**. El vector ocuparà el mateix espai independentment del nombre d'elements que emmagatzemi, per tant, es desaprofiten les posicions que no guarden elements. Aquest fet pot ser important quan un programa treballa amb diverses estructures de dades i arriba un moment en què el programa no pot continuar perquè una de les estructures s'ha omplert mentre la resta encara tenia espai lliure.

- S'ha de **determinar inicialment la capacitat del vector** sense disposar de prou informació. Generalment, quan es declaren les variables, no es té una idea clara de quants elements emmagatzemarà cada vector.

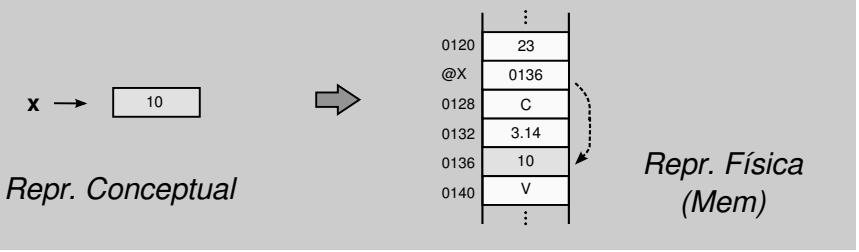
- Tal com hem vist en les implementacions fetes, el **nostre codi ha hagut de gestionar l'espai lliure** en el vector.

4.2. L'alternativa: punters

L'alternativa a l'ús dels vectors és la **gestió de memòria dinàmica**. Aquest mecanisme ens evita haver d'estimar inicialment l'espai que necessitarem per a emmagatzemar els elements, ja que demanarem l'espai a mesura que el necessitem.

Per a treballar amb la memòria dinàmica, utilitzarem un tipus anomenat **punter**, de manera que una variable de tipus `punter a T` (en què *T* és un tipus de dades concret) serà un apuntador a un objecte de tipus *T*. El punter oferirà un camí d'accés a l'objecte apuntat. Per exemple, si `x` és una variable de tipus `punter a Enter`, llavors `x` apuntarà a un enter (figura 21).

Figura 21. Exemple d'un punter a enter



Un **punter** és una variable que guarda l'adreça de memòria on comença a emmagatzemar-se l'objecte al qual apunta.

En aquest cas, el sistema operatiu és l'encarregat de gestionar l'espai de memòria. Per a treballar amb els punters disposem de les **primitives** i **operadors** següents:

- **punter a**: serveix per a declarar un tipus d'apuntadors a objectes d'un altre tipus.

  Exemple

  Declaració d'un tipus `tpAbc` d'apuntadors a objectes de tipus `tAbc`.

  ```
  tipus
    tAbc = tupla
             e : enter;
             ...
    ftupla
    tpAbc = punter a tAbc;
  ftipus
  ```

- **obtenirEspai**: funció per a sol·licitar l'espai necessari per a emmagatzemar un objecte del tipus apuntat. La funció retorna un punter que apunta a l'espai concedit per a emmagatzemar l'objecte.

  Exemple

  El punter `p` apunta a un objecte del tipus `tAbc`.

  ```
  var p : tpAbc fvar
    p := obtenirEspai();
  ```

- **operador '^'**: operador que permet accedir a l'objecte apuntat per un punter. Escriurem el símbol darrera del nom del punter i retornarà l'objecte apuntat pel punter.

  Exemple

  S'assigna 8 al camp `e` de la tupla apuntada pel punter `p`.

  ```
  p^.e := 8;
  ```

- **alliberarEspai**: acció que destrueix l'objecte apuntat pel punter que rep com a paràmetre. L'objecte deixa de ser accessible i l'espai ocupat és alliberat per a poder ser reutilitzat.

  Exemple

  S'allibera l'objecte de tipus `tAbc` apuntat pel punter `p`.

  alliberarEspai(*p*);

- **NUL**: és un valor especial que indica que un punter no apunta enlloc. La funció `obtenirEspai` retorna aquest valor quan no queda espai disponible i, en conseqüència, no pot proporcionar l'espai sol·licitat. L'acció `alliberarEspai` assigna el valor `NUL` al punter un cop ha destruït l'objecte apuntat. D'altra banda, obtindrem un error sempre que cridem l'acció `alliberarEspai` o apliquem l'operador '^' sobre un punter que contingui el valor `NUL`.

  Exemple

  Després de sol·licitar l'espai per a emmagatzemar un objecte, sempre s'ha de comprovar si el sistema ens l'ha concedit verificant que el punter `p` apunta a algun lloc.

```
var p : tpAbc fvar
  p := obtenirEspai();
  si p = NUL llavors error {no hi ha espai lliure};
  sino
       p^.e := p^.e + 1;
  fsi
```

4.3. Implementació

Tornem a implementar les estructures bàsiques explicades anteriorment, però en lloc d'utilitzar vectors aquest cop utilitzarem punters.

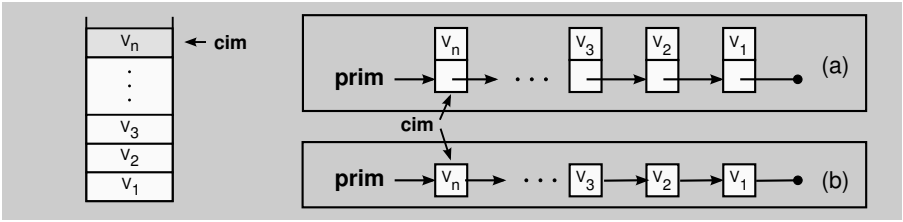
Fixeu-vos que ja no s'utilitzarà la constant que indicava la capacitat màxima de l'estructura, però sí que es continuarà controlant que hi hagi prou espai lliure en la inserció de nous elements.

L'encadenament entre elements (`suc`) es farà amb punters.

4.3.1. Pila

Tot seguit establim la representació i us mostrem la implementació de les operacions del tipus pila. De la representació (figura 22), comentarem que el punter `prim` sempre apunta al **cim** de la pila, excepte quan la pila és buida.

Figura 22. Implementació d'una pila amb punters. D'ara endavant, utilitzarem l'esquema simplificat (b) per a dibuixar les estructures amb punters.



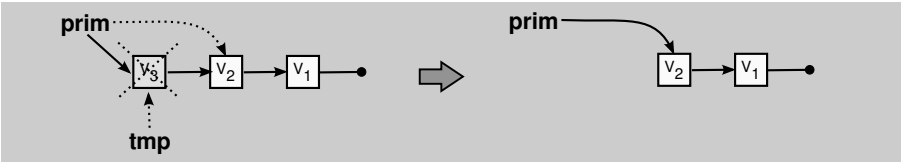

```
tipus
   node = tupla
             e : elem;
             suc : punter a node;
   ftupla
   pila = tupla
             prim : punter a node;
   ftupla
ftipus
```

Els **nodes** són els components emprats en la construcció de les estructures de dades quan es treballa amb punters. Cada **node** conté l'element a emmagatzemar i un o més encadenaments a altres nodes per a poder-se desplaçar entre ells.

```
funcio crear() : pila
   var p : pila fvar
   p.prim := NUL;
   retorna p;
ffuncio

funcio empilar(p : pila; e : elem) : pila
   var tmp : punter a node fvar
   tmp := obtenirEspai();
   si tmp = NUL llavors error {no hi ha espai lliure};
   sino
         tmp^.e := e;
         tmp^.suc := p.prim;
         p.prim := tmp;
   fsi
   retorna p;
ffuncio
```

```
funcio desempilar(p : pila) : pila
  var tmp : punter a node fvar
  si p.prim = NUL llavors error {pila buida};
  sino
        tmp := p.prim;
        p.prim := p.prim^.suc;
        alliberarEspai(tmp);
  fsi
  retorna p;
ffuncio
```

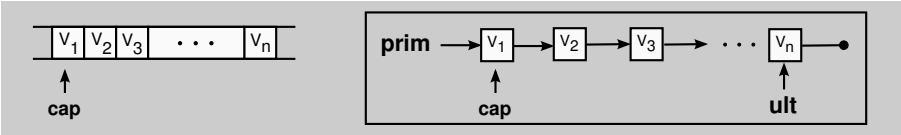
Figura 23. Operació `desempilar`




```
funcio cim(p : pila) : elem
  var e : elem fvar
  si p.prim = NUL llavors
       error {pila buida};
  sino
        e := p.prim^.e;
  fsi
  retorna e;
ffuncio

funcio buida(p : pila) : boolea
  retorna p.prim = NUL;
ffuncio
```

4.3.2. Cua

Figura 24. Implementació d'una cua amb punters



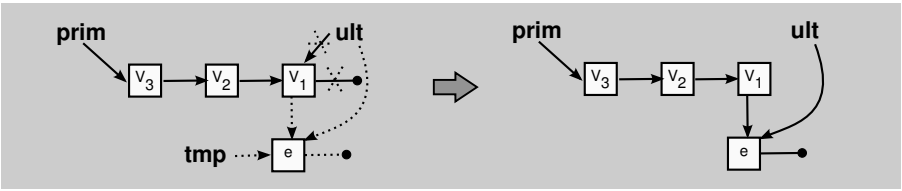
El **principi** de la cua (o extrem per on es treuen els elements) estarà sempre apuntat pel punter `prim`, mentre que el **final** de la cua (o l'extrem per on s'afegeixen els elements) estarà apuntat pel punter `ult` (figura 24). Quan la cua estigui buida, els punters `prim` i `ult` no apuntaran a cap element.

```
tipus
  node = tupla
           e : elem;
           suc : punter a node;
  ftupla
  cua = tupla
          prim,ult : punter a node;
  ftupla
ftipus
```

Donada la representació del tipus cua, mostrem la implementació de les seves operacions:

```
funcio crear() : cua
  var c : cua fvar
  c.prim := NUL;
  c.ult := NUL;
  retorna c;
ffuncio

funcio encuar(c : cua; e : elem) : cua
  var tmp : punter a node fvar
  tmp := obtenirEspai();
  si tmp = NUL llavors error {no hi ha espai lliure};
  sino
       tmp^.e := e;
       tmp^.suc := NUL;
       c.ult^.suc := tmp;
       c.ult := tmp;
  fsi
  retorna c;
ffuncio
```

Figura 25. Operació encuar

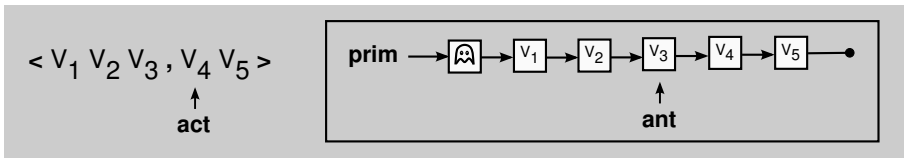


```
funcio desencuar(c : cua) : cua
  var tmp : punter a node fvar
  si c.prim = NUL llavors error {cua buida};
  sino
        tmp := c.prim;
        c.prim := c.prim^.suc;
        alliberarEspai(tmp);
  fsi
  retorna c;
ffuncio

funcio cap(c : cua) : elem
  var e : elem fvar
  si c.prim = NUL llavors error {cua buida};
  sino
        e := c.prim^.e;
  fsi
  retorna e;
ffuncio

funcio buida(c : cua) : boolea
  retorna c.prim = NUL;
ffuncio
```

4.3.3. Llista

Tornem a triar la **llista encadenada amb punt d'interès i element fantasma**. El punter `ant` apuntarà a l'element anterior del punt d'interès i no a l'element distingit. D'aquesta manera es podrà accedir fàcilment al predecessor de l'element distingit (figura 26).

Figura 26. Implementació d'una llista encadenada amb punters

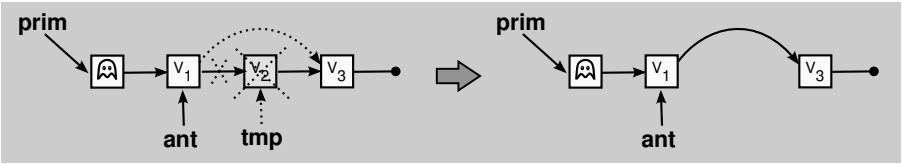



```
tipus
  node = tupla
            e : elem;
            suc : punter a node;
  ftupla
  llista = tupla
            prim,ant : punter a node;
  ftupla
ftipus
```

Donada la representació del tipus llista, mostrem la implementació de les seves operacions:

```
funcio crear() : llista
   var l : llista fvar
   l.prim := obtenirEspai();     {per l'element fantasma}
   si l.prim = NUL llavors
         error {no hi ha espai lliure};
   sino
         l.ant := l.prim;
         l.ant^.suc := NUL;
   fsi
   retorna l;
ffuncio

funcio inserir(l : llista; e : elem) : llista
   var tmp : punter a node fvar
   tmp := obtenirEspai();
   si tmp = NUL llavors
         error {no hi ha espai lliure};
   sino
         tmp^.e := e;
         tmp^.suc := l.ant^.suc;
         l.ant^.suc := tmp;
         l.ant := tmp;
   fsi
   retorna l;
ffuncio

funcio esborrar(l : llista) : llista
   var tmp : punter a node fvar
   si l.ant^.suc = NUL llavors
         error {fi de llista o llista buida};
   sino
         tmp := l.ant^.suc;
         l.ant^.suc := tmp^.suc;
         alliberarEspai(tmp);
   fsi
   retorna l;
ffuncio
```

Figura 27. Operació `esborrar` en una llista



```
funcio principi(l : llista) : llista
  l.ant := l.prim;
  retorna l;
ffuncio

funcio avançar(l : llista) : llista
  si l.ant^.suc = NUL llavors
      error {fi de llista o llista buida};
  sino
      l.ant := l.ant^.suc;
  fsi
  retorna l;
ffuncio

funcio actual(l : llista) : elem
  var e : elem fvar
  si l.ant^.suc = NUL llavors
      error {fi de llista o llista buida};
  sino
      e := l.ant^.suc^.e;
  fsi
  retorna e;
ffuncio

funcio fi(l : llista) : boolea
  retorna l.ant^.suc = NUL;
ffuncio

funcio buida(l : llista) : boolea
  retorna l.prim^.suc = NUL;
ffuncio
```

4.4. Perills dels punters

Abans de continuar, cal tenir en compte que els punters no són la panacea per a implementar estructures de dades, ja que també presenten problemes. Als següents subapartats en veurem uns quants.

4.4.1. La memòria dinàmica no és infinita

La memòria dinàmica no pot garantir la capacitat d'una estructura (quants objectes hi cabran), perquè és un recurs compartit (com un sac) en què tots els programes sol·liciten l'espai que necessiten. Això impossibilita que el gestor de memòria pugui predir totes les peticions que faran els programes i, per tant, desconegui l'espai de què disposarà en cada instant.

Per aquest motiu, sempre que se sol·licita espai a la memòria dinàmica, hem de comprovar que se'ns ha concedit, perquè en qualsevol moment l'espai lliure es pot esgotar.

De totes maneres, si necessiteu assegurar una capacitat màxima en una estructura, llavors probablement haureu d'implementar l'estructura amb vectors, ja que la mida declarada inicialment està garantida per a tota l'execució.

4.4.2. El funcionament del TAD depèn de la representació triada

Aquest punt és inacceptable, ja que vol dir que el comportament de l'algorisme dependrà de la implementació que s'hagi fet del tipus. I això vol dir que es **perd la propietat principal dels TAD, que és l'abstracció**, en altres paraules, es perd la transparència alhora d'utilitzar els tipus.

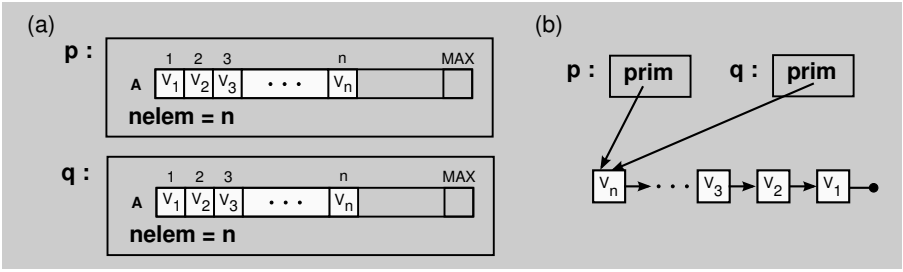
A continuació, us detallem amb exemples quines construccions algorísmiques es comporten diferent depenent de si el tipus s'ha implementat amb vectors o amb punters, i també us detallem com solucionar-ho:

1) Assignació. Donades dues variables `p` i `q` de tipus pila, el resultat de l'assignació `p := q` depèn de com s'hagi implementat el tipus (figura 28):

a) Si s'han utilitzat vectors, llavors `p` i `q` són dues tuples que contenen un vector i un enter. L'assignació copiarà el camp enter i el contingut del vector, donant com a resultat **dues piles iguals i independents**.

b) Si s'han utilitzat punters, llavors `p` i `q` són dues tuples que contenen un punter (`prim`). L'assignació simplement copiarà el camp de les tuples, és a dir, el punter `prim` de les dues piles apuntarà a la mateixa adreça de memòria. Per tant, el resultat serà una **única pila** compartida per `p` i `q`.

Figura 28. Resultat de l'operació `p := q` depenent de la implementació triada per les piles



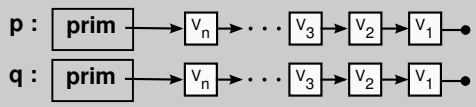
La solució consistirà a definir una nova operació anomenada **duplicar** que farà la còpia exacta d'un objecte sobre un altre objecte del mateix TAD.

Com a exemple, us mostrem la implementació de l'operació `duplicar` en el TAD *pila*, tant per a la representació amb vectors com per a la de punters:

```
funcio duplicar(p : pila) : pila
  var q : pila; i : enter fvar
  q.nelem := p.nelem;
  per i := 1 fins p.nelem fer
      q.A[i] := duplicar(p.A[i]);
  fper
  retorna q;
ffuncio
```

```
funcio duplicar(p : pila) : pila
  var q : pila; n1,n2 : punter a node fvar
  si p.prim = NUL llavors
      q.prim = NUL;
  sino
      n1 := p.prim;
      n2 := obtenirEspai();
      si n2 = NUL llavors
          error {no hi ha espai lliure};
      sino
          q.prim := n2;
          n2^.e := duplicar(n1^.e);
      fsi
      mentre n1^.suc ≠ NUL fer
          n2^.suc := obtenirEspai();
          si n2^.suc = NUL llavors
              error {no hi ha espai lliure};
          sino
              n1 := n1^.suc;
              n2 := n2^.suc;
              n2^.e := duplicar(n1^.e);
          fsi
      fmentre
      n2^.suc := NUL;
  fsi
  retorna q;
ffuncio
```

Figura 29. Resultat de l'operació `duplicar(p,q)`, implementant les piles amb punters



2) Comparació. Donades dues variables `p` i `q` de tipus pila, el resultat de la comparació `p = q` torna a dependre de com s'hagi implementat el tipus:

a) Si s'han utilitzat vectors, llavors `p` i `q` són dues tuples que contenen un vector i un enter. L'operador de comparació comparará el valor dels camps

i del contingut del vector, i donarà com a resultat la comparació d'igualtat correcta.

b) Si s'han utilitzat punters, llavors `p` i `q` són dues tuples que contenen un punter (`prim`). L'operador de comparació compararà el valor dels camps, és a dir, compararà si el punter `prim` de les dues piles apunta a la mateixa adreça de memòria. Per tant, el **resultat no serà una comparació del contingut de les piles `p` i `q`**, sinó una comparació de la seva posició de memòria.

> Per resoldre aquest problema aplicarem la mateixa solució que en l'operació assignació. En lloc d'utilitzar l'operador d'igualtat, cada TAD definirà una operació anomenada **igual** que comprovarà si dos objectes són iguals segons la definició d'igualtat en el tipus.

Per exemple, en el cas de les piles direm que dues piles són iguals si contenen els mateixos elements en el mateix ordre, mentre que el cas de les llistes també demanarem que el punt d'interès estigui situat en el mateix lloc.

Com a exemple de l'operació `igual`, us adjuntem la seva implementació en el TAD *pila*, tant per a la representació amb vectors (primera funció) com per a la de punters (segona funció):

```
funcio igual(p1,p2 : pila) : boolea
  var ig : boolea; i : enter fvar
  ig := (p1.nelem = p2.nelem);
  i := 1;
  mentre (i ≤ p.nelem) ∧ ig fer
          ig := igual(p1.A[i],p2.A[i]);
          i := i + 1;
  fmentre
  retorna ig;
ffuncio

funcio igual(p1,p2 : pila) : boolea
  var ig : boolea; n1,n2 : punter a node fvar
  n1 := p1.prim;
  n2 := p2.prim;
  ig := CERT;
  mentre (n1 ≠ NUL) ∧ (n2 ≠ NUL) ∧ ig fer
          ig := igual(n1^.e,n2^.e);
          si ig llavors
              n1 := n1^.suc;
              n2 := n2^.suc;
          fsi
  fmentre
  retorna (n1 = NUL) ∧ (n2 = NUL);
ffuncio
```

3) Paràmetres d'entrada. Donada una funció que declara un punter com a paràmetre d'entrada, tenim que **només està protegit el valor del punter, però en canvi no ho està el valor de l'objecte apuntat pel punter**. Això implica que si passem una pila `p` com a paràmetre d'entrada d'una funció, la protecció del contingut dependrà de com s'hagi implementat el tipus:

a) Si s'han utilitzat vectors, llavors `p` conté un vector i un enter. En aquest cas, els camps es dupliquen i cap dels camps de la tupla `p` estarà modificat en tornar de la crida a la funció.

b) Si s'han utilitzat punters, llavors `p` conté un punter al primer node de la pila (`prim`). El punter `prim` es duplicarà i en tornar de la crida a la funció no haurà estat modificat, però sí que es poden haver modificat els objectes emmagatzemats a la pila, ja que no estaven duplicats en no estar protegits per la definició de paràmetre d'entrada.

Per a solucionar aquest problema, simplement hem d'evitar modificar els paràmetre d'entrada. Si és necessari modificar algun paràmetre d'entrada en la funció, llavors utilitzarem l'operació `duplicar` del tipus per a crear una còpia de l'objecte i treballar-hi.

Però aquesta solució pot provocar un **problema d'ineficiència**, ja que la duplicació d'objectes pot ser molt costosa (per exemple, una pila plena d'elements). Per a resoldre-ho, podem convertir les funcions en accions, de manera que el resultat de l'operació quedi en un dels paràmetres d'entsor, i així ens estalviem el cost de la còpia.

Com a exemple de la problemàtica, us donem la implementació d'una funció per a fusionar dues piles sense mantenir l'ordre dels elements. Com que la funció és externa al tipus, no podrà accedir a la representació i només podrà utilitzar les operacions proporcionades pel TAD *pila*.

La primera versió és incorrecta si les piles s'han implementat amb punters perquè, en tornar de la crida, la pila `p2` s'ha buidat i la pila `p1` conté nous elements:

```
funcio fusionar(p1,p2 : pila) : pila
  mentre ¬buida(p2) fer
          p1 := empilar(p1,cim(p2));
          p2 := desempilar(p2);
  fmentre
  retorna p1;
ffuncio
```

La segona implementació és correcta, ja que és independent de la implementació perquè crea unes còpies (`pa` i `pb`) per protegir les piles `p1` i `p2`:

```
funcio fusionar(p1,p2 : pila) : pila
  var pa,pb : pila fvar
  pa := duplicar(p1);
  pb := duplicar(p2);
  mentre ¬buida(pb) fer
            pa := empilar(pa,cim(pb));
            pb := desempilar(pb);
  fmentre
  retorna pa;
ffuncio
```

Per acabar, la tercera implementació és la més eficient de les tres, ja que evita la duplicació de la pila `p1` convertint-la en un paràmetre d'*entsor*:

```
accio fusionar(entsor p1 : pila; ent p2 : pila)
  var pb : pila fvar
  pb := duplicar(p2);
  mentre ¬buida(pb) fer
            p1 := empilar(p1,cim(pb));
            pb := desempilar(pb);
  fmentre
faccio
```

4.4.3. Efectes laterals

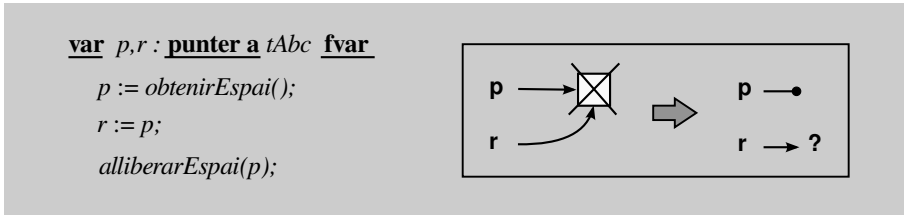
Donat un objecte apuntat per diversos punters, qualsevol modificació de l'objecte mitjançant un punter comporta un **efecte lateral**, i és que la resta de punters veuran l'objecte modificat sense haver-lo tocat.

Hi ha efectes laterals desitjats o controlats, però també n'hi ha d'erronis o no controlats.

4.4.4. Referències penjades

Parlem de **referència penjada** quan un punter apunta a un objecte que no existeix. Si provem d'accedir-hi (amb l'operador ^) el resultat és impredictible. Per exemple, en la figura 30, el punter `r` acaba penjat.

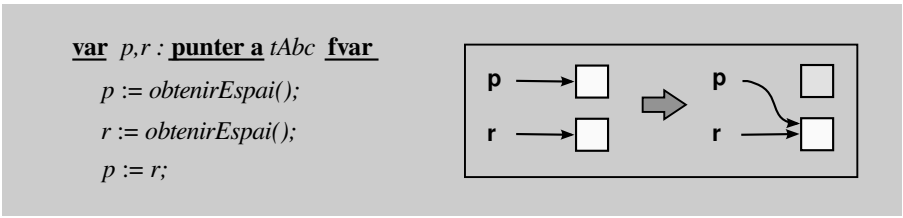
Figura 30. Exemple d'una referència penjada



4.4.5. Retalls

Un **retall** és un objecte que obté l'espai mitjançant la funció `obtenirEspai`, però que és inaccessible perquè no està apuntat per cap punter. No hi ha cap manera d'accedir-hi. La figura 31 mostra com l'objecte apuntat inicialment pel punter `p` es converteix en un retall.

Figura 31. Exemple d'un retall



Els retalls poden provocar un error del sistema operatiu o simplement no afectar l'execució i dificultar la depuració del codi. Hi ha sistemes operatius que s'encarreguen de recuperar automàticament aquests retalls o objectes inaccessibles, però mirarem d'evitar-los sempre que sigui possible, ja que són perillosos.

Alguns exemples de situacions que poden crear retalls i hem de vigilar són:

- La **declaració de variables auxiliars** del tipus. Si els objectes s'han implementat amb punters, en sortir d'una funció quedaran com a retalls, ja que s'eliminaran els punters que els feien accessibles.

```
accio concatenar(entsor l1 : llista; ent l2 : llista)
  var lb : llista fvar
  ...
  lb := duplicar(l2);
  ...
  {buida(lb) = FALS}
faccio
```

- Un altre exemple el tenim en el que anomenarem **reinicialització de variables**. En un punt del programa, el programador pot decidir reutilitzar una variable (que ja no s'utilitza) per a no haver de declarar-ne més. Si la inicialitza cridant l'operació de creació (**crear**) llavors generarà un retall, perquè l'objecte anterior a la creació serà inaccessible sense haver alliberat el seu espai.

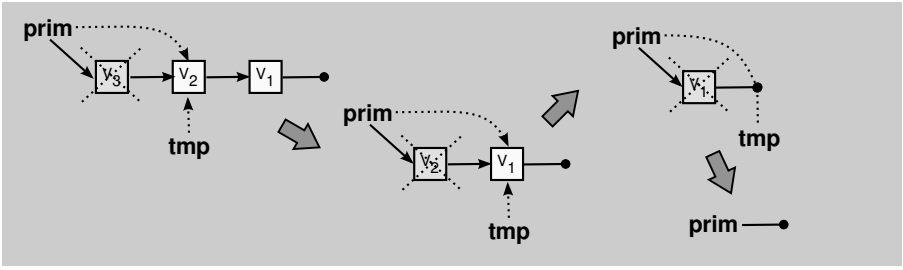
```
var p : pila fvar
p := crear();
...
p := empilar(p,e);
...
{buida(p) = FALS}
p := crear();
...
```

Per solucionar aquest problema, afegim una operació anomenada `destruir` al TAD, que s'encarregarà d'alliberar l'espai ocupat per un objecte.

Com a exemple us adjuntem la implementació de l'operació `destruir` en el TAD *pila*, tant per a la representació amb vectors com per a la de punters:

```
accio destruir(entsor p : pila)
  var i : enter fvar
  per i := 1 fins p.nelem fer
      destruir(p.A[i]);
  fper
faccio
```

Figura 32. Resultat de l'operació `destruir(p)`, si la pila p s'ha implementat amb punters.




```
accio destruir(entsor p : pila)
  var tmp : punter a node fvar
  mentre p.prim ≠ NUL fer
        tmp := p.prim^.suc;
        destruir(p.prim^.e);
        alliberarEspai(p.prim);
        p.prim := tmp;
  fmentre
faccio
```

Finalment, com a exemple, implementem una acció que concatena dues llistes i que inclou una crida a la nova operació `destruir` per a alliberar l'espai que ocupa la variable auxiliar `lb` i així evitar el problema dels retalls:

```
accio concatenar(entsor l1 : llista; ent l2 : llista)
  var lb : llista fvar
  mentre ¬fi(l1) fer
          l1 := avançar(l1);
  fmentre
  lb := duplicar(l2);
  lb := principi(lb);
  mentre ¬fi(lb) fer
          l1 := inserir(l1,actual(lb));
          lb := avançar(lb);
  fmentre
  destruir(lb);
faccio
```

4.4.6. Conclusió

Els punters ofereixen més possibilitats i flexibilitat a l'hora de dissenyar estructures, però també s'han d'utilitzar amb més cura, ja que són una font més gran de problemes.

Així mateix, sempre que implementem un TAD inclourem:

- Les operacions indicades en la signatura del tipus, perquè són les que proporcionen la funcionalitat del tipus.

- Les operacions `igual`, `duplicar` i `destruir` perquè fan el tipus transparent a la implementació.

Resumint, la implementació d'un TAD ha d'incloure les operacions de la signatura i les operacions `igual`, `duplicar` i `destruir`.

4.5. Exemple d'implementació definitiva: Llista encadenada

```
tipus
  node = tupla
           e : elem;
           suc : punter a node;
  ftupla
  llista = tupla
           prim,ant : punter a node;
  ftupla
ftipus
```

```
accio crear(sor l : llista)
  si l.prim ≠ NUL llavors
      destruir(l);     {per evitar retalls}
  fsi
  l.prim := obtenirEspai();     {per l'element fantasma}
  si l.prim = NUL llavors
      error {no hi ha espai lliure};
  sino
      l.ant := l.prim;
      l.ant^.suc := NUL;
  fsi
faccio

accio inserir(entsor l : llista; ent e : elem)
  var p : punter a node fvar
  p := obtenirEspai();
  si p = NUL llavors
      error {no hi ha espai lliure};
  sino
      duplicar(p^.e,e);
      p^.suc := l.ant^.suc;
      l.ant^.suc := p;
      l.ant := p;
  fsi
faccio

accio esborrar(entsor l : llista)
  var p : punter a node fvar
  si l.ant^.suc = NUL llavors
      error {fi de llista o llista buida};
  sino
      p := l.ant^.suc;
      l.ant^.suc := p^.suc;
      destruir(p^.e);     {per evitar retalls}
      alliberarEspai(p);
  fsi
faccio

accio principi(entsor l : llista)
  l.ant := l.prim;
faccio
```

```
accio avançar(entsor l : llista)
  si l.ant^.suc = NUL llavors
      error {fi de llista o llista buida};
  sino
      l.ant := l.ant^.suc;
  fsi
faccio

funcio actual(l : llista) : elem
  var e : elem fvar
  si l.ant^.suc = NUL llavors
      error {fi de llista o llista buida};
  sino
      duplicar(e,l.ant^.suc^.e);
  fsi
  retorna e;
ffuncio

funcio fi(l : llista) : boolea
  retorna l.ant^.suc = NUL;
ffuncio

funcio buida(l : llista) : boolea
  retorna l.prim^.suc = NUL;
ffuncio

accio duplicar(sor l2 : llista; ent l1 : llista)
  var n1,n2 : punter a node; e : elem fvar
  crear(l2);
  n1 := l1.prim^.suc;
  n2 := l2.prim;
  mentre n1 ≠ NUL fer
          n2^.suc := obtenirEspai();
          si n2^.suc = NUL llavors
              error {no hi ha espai lliure};
          sino
              n2 := n2^.suc;
              duplicar(n2^.e,n1^.e);
              si n1 = l1.ant llavors
                  l2.ant := n2;
              fsi
              n1 := n1^.suc;
          fsi
  fmentre
  n2^.suc := NUL;
faccio
```

```
funcio igual(l1,l2 : llista) : boolea
  var ig : boolea; n1,n2 : punter a node fvar
  n1 := l1.prim;
  n2 := l2.prim;
  ig := (igual(l1.ant,n1) ∧ igual(l2.ant,n2)) ∨ (¬igual(l1.ant,n1) ∧
        ¬igual(l2.ant,n2));     {No comparem el contingut dels elements fantasma}
  si ig llavors
        n1 := n1^.suc;
        n2 := n2^.suc;
  fsi
  mentre (n1 ≠ NUL) ∧ (n2 ≠ NUL) ∧ ig fer
              ig := igual(n1^.e,n2^.e) ∧ ((igual(l1.ant,n1) ∧ igual(l2.ant,n2)) ∨
                    (¬igual(l1.ant,n1) ∧ ¬igual(l2.ant,n2)));
              si ig llavors
                    n1 := n1^.suc;
                    n2 := n2^.suc;
              fsi
  fmentre
  retorna (n1 = NUL) ∧ (n2 = NUL);
ffuncio

accio destruir(entsor l : llista)
  var tmp : punter a node fvar
  mentre l.prim ≠ NUL fer
              tmp := l.prim^.suc;
              destruir(l.prim^.e);
              alliberarEspai(l.prim);
              l.prim := tmp;
  fmentre
faccio
```

4.6. Altres variants

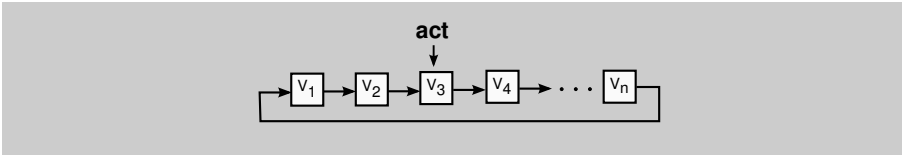
En l'àmbit del disseny d'estructures de dades complexes, les llistes encadenades poden presentar mancances i ineficiències. Per afrontar aquesta situació, introduïm breument tres variants de llistes: les llistes circulars, les listes doblement encadenades i les llistes ordenades.

4.6.1. Llistes circulars

La **llista circular** recicla l'encadenament de l'últim element perquè apunti al primer element de la llista. El resultat és una llista sense primer ni últim.

L'avantatge principal d'aquest tipus de llista és que donat un element sempre es pot accedir a qualsevol altre element de la llista, perquè l'últim element de la llista sempre apunta al primer (figura 33).

Figura 33. Llista circular



La llista circular facilita la implementació dels algorismes ja que, com que tots els encadenaments són del mateix tipus, ens evita haver de codificar un tractament especial per a l'últim element.

Si afegim un punter a la llista circular per a designar un element com el primer, llavors podrem variar el principi de la llista simplement desplaçant aquest punter, mentre que en la llista encadenada era necessari desplaçar elements.

Per acabar, podem implementar una cua amb una llista circular afegint un punter a l'últim element, ja que el primer element sempre serà el successor de l'últim.

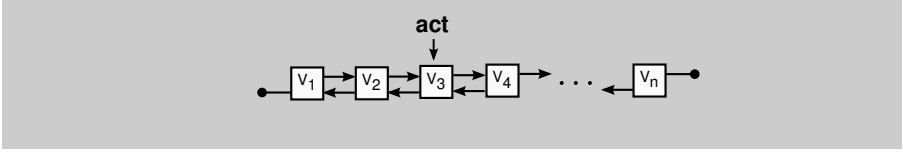
4.6.2. Llistes doblement encadenades

La llista encadenada només permet fer recorreguts en un sentit, ja que donat un element només en coneixem el successor. Això comporta que per a conèixer l'element anterior s'hagi de recórrer la llista des del principi i que el cost d'aquesta operació sigui lineal. Per tant, **si es vol recórrer la llista en els dos sentits (cap endavant i cap enrere) és necessari afegir encadenaments en totes dues direccions**.

Aquesta és la idea de la **llista doblement encadenada**, en què cada element té dos encadenaments: un al node anterior i l'altre al node següent (figura 34).

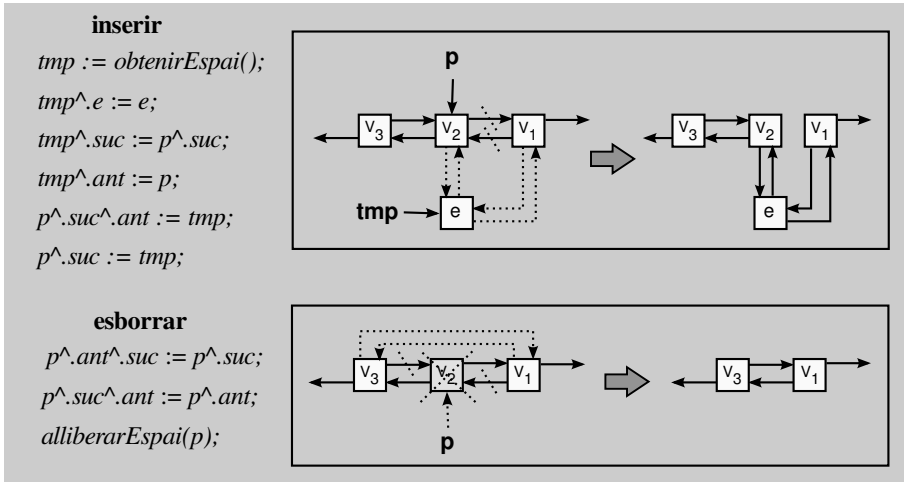
El cost de les operacions tornarà a ser constant, perquè l'accés a l'element predecessor serà immediat tot i consumir una mica més d'espai.

Figura 34. Llista doblement encadenada



Però el doble encadenament no solament és útil en els recorreguts, també ho és **quan la llista forma part d'una estructura de dades complexa i els seus elements són apuntats des d'altres punts de l'estructura**. Fruit d'aquesta relació, des d'altres punts de l'estructura es poden suprimir arbitràriament elements de la llista encadenada i llavors és quan resulta necessari accedir al predecessor de l'element suprimit per a actualitzar els encadenaments de la llista (figura 35).

Figura 35. Esquelet de les operacions d'inserció i esborrat d'un element de la llista



4.6.3. Llistes ordenades

Donada una llista qualsevol, de vegades és necessari fer un recorregut ordenat dels elements pel valor d'un dels seus camps. Aquest camp s'anomenarà `clau`.

Per a disposar d'una **llista ordenada** no s'ha de modificar la representació, sinó que només s'ha de modificar la implementació de certes operacions amb la finalitat següent:

- mantenir la llista desordenada en les actualitzacions i ordenar-la abans de començar un recorregut, o bé
- mantenir la llista sempre ordenada.

Si triem la primera opció, llavors s'haurà de modificar l'operació `principi`, que es convertirà en l'operació més costosa, ja que haurà d'ordenar tots els elements de la llista per a fer el recorregut. Mentre que si triem la segona alternativa, l'operació a modificar és `inserir`, ja que ha de cercar seqüencialment el punt on ha d'afegir l'element per a mantenir l'ordenació. En aquest cas, l'operació més costosa serà `inserir`, però a canvi la llista sempre estarà ordenada.

Depenent de com es distribueixin les insercions i els recorreguts en una execució, ens interessarà una alternativa o una altra. Per exemple, si l'algorisme

presenta dues fases diferenciades: una primera en què es construeix la llista i una segona en què només es consulta, llavors la primera alternativa sembla la millor, ja que les actualitzacions tindran el cost mínim i només s'ordenarà la llista un sol cop (just en el primer recorregut).

Resum

En aquest mòdul, s'han introduït els tipus bàsics de dades per a emmagatzemar les seqüències: *piles*, *cues* i *llistes*. Tots tres tipus ofereixen operacions per a accedir d'una manera seqüencial a les dades, però es diferencien en la manera de fer-ho. Així, doncs, depenent de la situació en què ens trobem, triarem el tipus que més s'adeqüi al comportament requerit.

També, hem vist com es treballa amb un tipus de dades a partir de la seva signatura sense conèixer la implementació. Aquesta és la característica principal d'un *tipus abstracte de dades*. Així mateix, hem comentat com s'afegeixen *funcions externes* al tipus i com s'integren diferents tipus en una estructura de dades complexa.

Hem après a construir nous TAD per a poder modelitzar situacions del món real. Primer, especificant la *signatura* del tipus amb les operacions que s'hi podran aplicar, i tot seguit abordant la seva *implementació*. Per a fer-ho, s'escollirà una representació de les dades i es codificaran les operacions avaluant-ne el cost espacial i temporal.

Aquesta avaluació és important perquè, donada l'especificació d'un tipus, sempre hi haurà més d'una implementació possible i necessitarem un sistema per a decidir quina és la implementació més adequada per al nostre problema.

A més, hem estudiat com s'emmagatzemen dades aprofitant que la representació era *seqüencial*, i hem vist alguns dels problemes que presenta aquesta alternativa. Per superar-los, hem introduït el concepte de representació *encadenada*, que consisteix a especificar en l'estructura de dades la relació que hi ha entre els elements.

Un altre tema en què s'ha aprofundit és en l'ús dels *vectors* i dels *punters*. S'han implementat els tipus bàsics amb els dos sistemes i se n'han comentat els avantatges i desavantatges. També, s'ha vist que és necessari afegir les operacions igual, duplicar i destruir a la implementació d'un tipus, si es vol garantir un comportament idèntic independentment de la implementació triada.

D'altra banda, aprofitant el tema dels punters, s'ha explicat el sistema de *gestió de la memòria dinàmica*, com també els perills que presenta (retalls, referències penjades, etc.).

Per acabar, hem vist breument tres variants del tipus llista que aporten solucions per a situacions especials: *llistes circulars*, *llistes doblement encadenades* i *llistes ordenades*.

Activitats

1. Implementeu dues piles en un sol vector, de manera que el cost temporal sigui $\Theta(1)$ i que el total d'elements emmagatzemats no sobrepassi la capacitat del vector.

2. El TAD *llista* és el més flexible dels explicats en el mòdul, fins el punt de considerar els TAD *cua* i *pila* com a casos particulars d'aquest. Per demostrar-ho, implementeu una pila i una cua utilitzant una llista encadenada amb punt d'interès.

3. Esteneu el TAD *pila* (implementat amb vectors) afegint les operacions següents:

- `base(p)`, que retorna l'element que fa més temps que està empilat en la pila `p`;
- `ndesempilar(p,n)`, que desempila `n` elements de la pila `p`. Si la pila conté menys de `n` elements, llavors desempilarà tots els que hi hagi.

4. Esteneu el TAD *cua* implementat amb punters, afegint les operacions següents:

- `num(c)`, que retorna el nombre d'elements que hi ha en la cua `c`;
- `ndesencuar(c,n)`, que desencua `n` elements de la cua `c`. Si no hi ha prou elements en la cua per desencuar, llavors es considerarà una situació errònia.

5. Donat el TAD *llista* implementat en el mòdul com una llista encadenada amb punters, afegiu les operacions següents:

- `llegir(l,i)`, que retorna l'`i`-èssim element situat a la dreta del punt d'interès de la llista `l`;
- `essim(l)`, que situa el punt d'interès sobre l'element `i`-èssim respecte al principi de la llista `l`. Si la llista és buida o no hi ha prou elements, llavors retorna el valor NUL.

6. En el mòdul, hem vist implementades les operacions `destruir`, `duplicar`, `igual` per a assegurar la transparència de la implementació del TAD *pila* i *llista*. Seguint l'exemple dels TAD comentats, implementeu aquestes operacions en el TAD *cua*.

7. Dissenyeu una acció que mostri la representació binaria d'un nombre llegit de la seqüència d'entrada. Per a fer-ho, podeu utilitzar una pila que emmagatzemi els resultats de les divisions parcials per la base (que és 2). Per exemple, si l'entrada és el número 77, llavors la sortida hauria de ser 1 0 0 1 1 0 1.

8. Construïu un nou TAD anomenat *picu* que barregi les operacions de les piles i de les cues sobre una seqüència: `crear, empilar, encuar, desencuar, desempilar, cim, cua` i `buida`.

9. Implementeu una cua utilitzant dues piles. Codifiqueu les operacions bàsiques de la signatura del tipus cua.

10. Creeu un TAD *conjunt* que representi un conjunt d'elements i que ofereixi com a mínim les operacions següents:

- `afegir(s,e)`: afegeix l'element `e` al conjunt `s`,
- `esborrar(s,e)`: esborra l'element `e` del conjunt `s`,
- `unio(s,t)`: afegeix els elements del conjunt `t` al conjunt `s`,
- `interseccio(s,t)`: esborra del conjunt `s` els elements que no són en el conjunt `t`,
- `diferencia(s,t)`: esborra del conjunt `s` els elements que són en el conjunt `t`,
- `mida(s)`: retorna el nombre d'elements del conjunt `s`.

11. Implementeu un TAD per modelitzar un editor de textos. Aquest TAD hauria de proporcionar les operacions següents:

- `afegir(t,c)`: afegeix el caràcter `c` en la posició del cursor i desplaça el cursor a la dreta,
- `esborrar(t)`: esborra el caràcter on està el cursor i desplaça el cursor al caràcter de l'esquerra,
- `esquerra(t)`: mou el cursor un caràcter a l'esquerra,
- `dreta(t)`: mou el cursor un caràcter a la dreta,
- `inici(t)`: mou el cursor a l'inici del text, o sobre el caràcter de més a l'esquerra,
- `cursor(t)`: retorna el caràcter sobre el que està el cursor.

12. Escriviu una acció `creixent`, que donada una llista d'enters, la retorni ordenada creixentment. Quina variació hauríeu de fer en el codi perquè l'ordenació de la llista fos estrictament creixent?

13. Una emissora de ràdio en línia vol automatitzar la gestió de les llistes de reproducció. Per aquest motiu, decideix organitzar les cançons de què disposa en una estructura de dades. Els directius de l'emissora decideixen que necessiten les operacions següents per a programar la música a emetre :

- `crear:` → `radio`, l'emissora té l'arxiu sonor completament buit.
- `afegir: radio canço` → `radio`, l'emissora afegeix una nova cançó al seu arxiu sonor.
- `suggerir: radio enter` → `canço`, retorna la cançó que té la durada donada en segons i que menys vegades ha estat emesa en la ràdio.
- `seleccionar: radio enter` → `canço`, retorna la cançó que té la durada donada en segons i que ha estat emesa més vegades.
- `emetre: radio canço` → `radio`, es pren nota que la cançó s'emetrà per la ràdio. Cada cop que s'emet una cançó es crida aquesta operació.
- `esborrar: radio enter` → `radio`, esborra les n-cançons que s'han emès menys vegades. En cas d'empat, és indiferent la cançó que se seleccioni.

El nombre de cançons de què disposa la ràdio és conegut, però també sabem que cada setmana l'emissora adquireix noves cançons i les afegeix a l'arxiu. Per cada cançó, s'emmagatzema un número que la identifica i la seva durada en segons. L'empresa us demana que definiu el TAD *ràdio* i l'implementeu de manera que totes les operacions tinguin el mínim cost temporal possible.

Exercicis d'autoavaluació

1. Dibuixeu el resultat d'aplicar la seqüència d'operacions següent sobre una pila buida `p`:
`empilar(p,1); empilar(p,7); desempilar(p); empilar(p,2); empilar(p,3); desempilar(p); empilar(p,5); empilar(p,9);`

2. Dibuixeu el resultat d'aplicar la seqüència d'operacions següent sobre una cua buida `q`:
`encuar(q,1); encuar(q,7); desencuar(q); encuar(q,2); encuar(q,3); desencuar(q); encuar(q,5); encuar(q,9);`

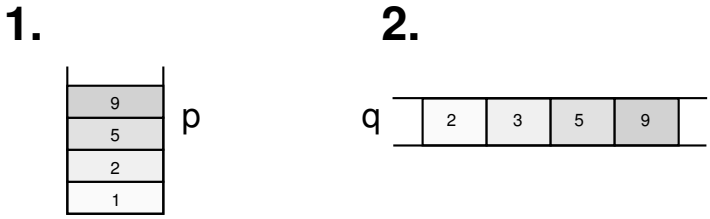
3. Escriviu una funció `balancejat` que llegeixi una seqüència d'entrada formada per parèntesis i claudàtors, i comprovi que tots estan correctament balancejats utilitzant una pila. És a dir, la funció ha de verificar que tots els parèntesis oberts es tanquen en l'ordre corresponent. La seqüència ha d'acabar amb un punt. Per exemple:

- Si l'entrada és "([ ] (( )) [ ] ." , llavors es retornarà CERT.
- En canvi, si l'entrada fos "([( )] ." , llavors es retornaria FALS.

4. Escriviu una acció `fibonacci` que calculi la sèrie de Fibonacci amb l'ajuda d'una cua. La sèrie de Fibonacci comença així: 0 1 1 2 3 5 8 13 21 34 55 89 ...

Solucionari

1. i 2.



3. Cada cop que l'algorisme llegeix un claudàtor o parèntesis d'obertura, ha d'empilar el símbol en la pila. Mentre que cada cop que troba un parèntesi o claudàtor de tancament en la seqüència d'entrada, ha de consultar el cim de la pila i comprovar que el símbol de tancament llegit es correspon amb el símbol d'obertura que hi ha al cim.

Si es correspon, llavors l'algorisme desempila el cim i continua l'execució. Altrament, si no coincideix o si la pila és buida, tindrem que l'expressió no estarà ben balancejada.

L'expressió estarà ben balancejada si en llegir el punt (.), que indica el final de la seqüència, la pila és buida.

```
funcio balancejat() : boolea
  var ok : boolea; c : caracter; p : pila(caracter) fvar
  crear(p);
  ok := CERT;
  c := llegirCaracter();
  mentre (c ≠' .') ∧ ok fer
          si (c ='(') ∨ (c ='[') llavors
               empilar(p,c);
               c := llegirCaracter();
          sino
               si ¬buida(p) llavors
                    si c =')' llavors ok := cim(p) ='(';
                                      sino ok := cim(p) ='[';
                    fsi
                    si ok llavors
                         desempilar(p);
                         c := llegirCaracter();
                    fsi
               sino
                    ok := FALS;
               fsi
          fsi
  fmentre
  retorna ok ∧ buida(p);
ffuncio
```

4.

```
accio fibonacci(ent n : enter)
  var q : cua(enter); a,b,i : enter; fvar
  crear(q);
  encuar(q,0);
  encuar(q,1);
  per i := 1 fins n fer
       a := cap(q);
       desencuar(q);
       b := cap(q);
       desencuar(q);
       encuar(q,b);
       encuar(q,a + b);
       escriureEnter(a);
  fper
faccio
```

Glossari

abstracció *f* Simplificació de la realitat tenint en compte només una part d'aquesta.

actual *adj* Dit de l'element distingit d'una llista que serveix de referència per a aplicar les operacions.

algorisme *m* Conjunt de passos per a resoldre un problema.

buit -da *adj* Que, aplicat a una estructura, no conté cap element.

cap *m* Element situat al principi d'una cua.

cim *m* Últim element arribat a una pila i que serà el primer a sortir.

cost constant $\Theta(1)$ *m* Mesura que és fixa i no depèn del nombre d'elements.

cost espacial *m* Mesura de l'espai que ocupa una implementació concreta d'un tipus de dades.

cost lineal $\Theta(n)$ *m* Mesura que es relaciona linealment amb el nombre d'elements.

cost temporal *m* Mesura del temps que requereix una operació en una implementació concreta d'un tipus de dades.

cua *f* TAD caracteritzat pel fet que el primer element a entrar és el primer a sortir.

encadenament *m* Indicador que identifica la posició que ocupa un altre element.

estat *m* Descripció d'una estructura en un instant determinat.

estructura circular *f* Estructura sense principi ni fi, on després de la darrera posició ve la primera posició.

fantasma *m* Element especial que es guarda en la primera posició d'una seqüència, perquè l'element actual sempre tingui un predecessor.

FIFO *f* *First in first out*.

funció externa *f* Funció que no pot accedir a la representació del tipus i només pot utilitzar les operacions facilitades per la signatura del tipus.

funció parcial *f* Funció que no està definida per a tot el domini dels paràmetres d'entrada.

implementació *f* Codificació concreta d'un tipus de dades.

LIFO *f* *Last in first out*.

llista *f* TAD caracteritzat pel fet que permet afegir, esborrar o consultar qualsevol element de la seqüència.

llista amb punt d'interès *f* Llista que conté un element distingit apuntat pel punt d'interès i que serveix de referència per a aplicar les operacions.

llista circular *f* Llista en què es recicla l'encadenament de l'últim element perquè apunti al primer element. El resultat és una llista sense primer ni últim.

llista doblement encadenada *f* Llista en què cada element té dos encadenaments: un al node anterior i l'altre al node següent.

llista ordenada *f* Llista en què els elements estan ordenats pel valor d'un dels camps. Aquest camp s'anomena *clau*.

memòria dinàmica *f* Mecanisme per a sol·licitar memòria en temps d'execució al sistema operatiu.

mòdul *m* Operació que calcula el residu de la divisió de dos nombres enters.

NUL *m* Valor especial que indica que un punter no apunta enlloc.

operació constructora *f* Operació del tipus que retorna un valor del mateix tipus.

operació consultora *f* Operació del tipus que retorna un valor d'un altre tipus.

operació generadora *f* Operació que forma part del conjunt mínim d'operacions constructores necessàries per a generar qualsevol valor del tipus.

operació modificadora *f* Operació constructora que no és generadora del tipus.

pila *f* TAD caracteritzat pel fet que l'últim element a entrar és el primer a sortir.

punter *f* Variable que emmagatzema l'adreça de memòria en què comença l'objecte al qual apunta.

referència penjada *f* Punter que apunta a un objecte que ja no existeix. Si provem d'accedir-hi el resultat és impredictible.

retall *m* Objecte que és inaccessible, perquè no està apuntat per cap punter, i no hi ha cap manera d'accedir-hi.

seqüència *f* Conjunt d'elements disposats en un ordre específic. Fruit d'aquesta ordenació, donat un element de la seqüència parlarem del predecessor (com l'element anterior) i del successor (com l'element següent).

signatura *f* Especificació formal del comportament de les operacions d'un tipus. Per cada operació estableix els paràmetres, el resultat, les condicions d'error i les equacions que en reflecteixen el comportament.

tipus abstracte de dades *m* Tipus de dades al que se li ha afegit el concepte abstracció per indicar que la implementació del tipus és invisible pels usuaris del tipus. Sigla **TAD**.

tipus de dades *m* Conjunt de valors i d'una sèrie d'operacions que s'hi poden aplicar. Les operacions compliran certes propietats que en determinaran el comportament.

vector *m* Taula unidimensional.

Bibliografia

Balcázar, J. L. (2001). *Programación metódica*. Madrid: McGraw-Hill.

Franch, X. (2001). *Estructures de dades. Especificació, disseny i implementació*. Barcelona: Edicions UPC.

Martí, N.; Ortega, Y.; Verdejo, J. A. (2001). *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*. Madrid: Pearson Educación.

Weiss, M. A (1995). *Estructuras de datos y algoritmos*. Madrid: Addison-Wesley.