Dades lineals en memòria dinàmica. 1

Isidre Guixà i Miranda

Estructures de dades dinàmiques

Maig del 2009
© Isidre Guixà i Miranda
IES SEP Milà i Fontanals
C/. Emili Vallès, 4
08700 - Igualada

En cas de suggeriment i/o detecció d'error podeu posar-vos en contacte via el correu electrònic `iguixa@xtec.cat`

Cap part d'aquesta publicació, incloent-hi el disseny general i de la coberta, no pot ser copiada, reproduïda, emmagatzemada o tramesa de cap manera ni per cap mitjà, tant si és elèctric, com químic, mecànic, òptic, d'enregistrament, de fotocòpia, o per altres mètodes, sense l'autorització prèvia per escrit dels titulars del copyright.

Índex

Introducció

En l'aprenentatge de la programació estructurada i modular hem après a gestionar dades de diversos tipus, uns facilitats pel llenguatge de programació i altres dissenyats per nosaltres mateixos. A més, aquesta gestió l'hem efectuat en memòria i també hem après com enregistrar les dades en suport extern amb la utilització dels sistemes gestors de fitxers.

Si pensem una mica en com desenvolupem un programa, ens adonarem que en el moment de dissenyar els algorismes, hem de fixar les variables a utilitzar i llur grandària. Això provoca problemes, ja que de vegades seria convenient decidir la gestió de la memòria en temps d'execució perquè d'aquesta manera es podria adequar a les necessitats de cada moment. Aquesta possibilitat, desconeguda per nosaltres, ens és facilitada amb l'aparició de dades dinàmiques.

No tots els llenguatges de programació faciliten la utilització de dades dinàmiques. En aquesta unitat didàctica, establirem els fonaments per a poder utilitzar dades dinàmiques en llenguatges que les suportin. El llenguatge C suporta dades dinàmiques i la seva gestió passa per la utilització correcta dels punters.

Així doncs, en el nucli d'activitat "Assignació dinàmica de memòria. Mecanismes en C", introduïm les operacions que els llenguatges acostumen a facilitar per a la gestió dinàmica de la memòria i, en especial, com cal actuar en utilitzar el llenguatge C, el qual basa tota la gestió dinàmica de la memòria en la correcta utilització dels punters. En l'aprenentatge de la programació en C, s'introdueixen els punters com a recurs imprescindible en el pas per referència en la crida de funcions. Us cal refrescar els coneixements introductoris sobre els punters en C.

Una vegada coneguts els mecanismes per assolir l'assignació dinàmica de la memòria, cal aplicar-los per a una eficient gestió de les dades en memòria i aquesta gestió passa per conèixer les principals estructures de dades dinàmiques emprades en la programació informàtica: llistes, piles, cues, arbres,...

En el nucli d'activitat "Llistes" presentarem l'estructura de dada dinàmica llista emprada quan cal emmagatzemar informació de manera lineal (element darrera element) i els principals algorismes per a la seva gestió: recorregut, inserció, recerca i eliminació.

En el nucli d'activitat "Piles i cues" presentarem aquestes estructures de dades dinàmiques, de les que ja podem avançar que no són altra cosa que tipus específics de llistes però que per la seva important contribució al món de la informàtica es mereixen un capítol específic.

Per aconseguir els nostres objectius, heu de reproduir en el vostre ordinador i, a ser possible, en diverses plataformes, tots els exemples incorporats en el text, per a la qual cosa, en la secció “Recursos de contingut”, trobareu tots els fitxers necessaris, a més de les activitats i els exercicis d’autoavaluació.

I un consell final: com que l’assignació dinàmica de memòria és una tècnica molt potent, ens cal conèixer-la i aplicar-la, però també és complicada i, per tant, cal anar amb molt de compte. Penseu que sereu vosaltres els gestors de la memòria de l’ordinador, passant per damunt del sistema operatiu. Per tant, si no ho fem de la manera adequada podem provocar el mal funcionament de l’ordinador. Evidentment, reiniciant la màquina tot queda solucionat. Però, i les dades que s’estaven gestionant..., i el programa que s’estava executant...? Molt de compte!

Objectius

A l'acabament d'aquesta unitat didàctica, l'estudiant ha de ser capaç de:

1. Utilitzar de manera correcta els punters en el llenguatge C.

2. Utilitzar, quan correspongui, memòria dinàmica en el disseny d'algorismes.

3. Dissenyar algorismes per a gestionar, de manera eficient, taules dinàmiques, llistes, piles i cues.

4. Identificar en quines situacions és convenient la utilització de taules dinàmiques, llistes, piles i cues.

1. Assignació dinàmica de memòria. Mecanismes en C.

La majoria d'aplicacions informàtiques gestionen dades que poden o no estar emmagatzemades en suport extern, però que mentre es gestionen, resideixen en la memòria de l'ordinador.

Els llenguatges de programació faciliten mecanismes per a que el programador pugui declarar dades en memòria. Així, per exemple, estem acostumats a declaracions del tipus:

- En pseudocodi:

```
var
   edat: natural;
   sou: real;
   dni: cadena[10];
   noms: taula[MAX_NOMS] de cadena[MAX_LONG_NOM];
fivar
```

- En llenguatge C:

```
unsigned int edat;
float sou;
char dni[10];
char noms[MAX_NOMS] [MAX_LONG_NOM];
```

Les declaracions anteriors, siguin en pseudocodi o en llenguatge C, tenen en comú que són declaracions de dades estàtiques. Recordem que:

> Les dades estàtiques existeixen durant tota l'execució del programa i/o acció/funció on estan declarades i tenen una capacitat fixa, mentre que les dades dinàmiques es poden crear i destruir en temps d'execució, segons evolucioni el programa, i poden acomodar la seva capacitat a les necessitats reals de l'execució.

La majoria de llenguatges de programació faciliten al programador mecanismes per a la gestió de dades dinàmiques. El llenguatge C no n'és una excepció i per a gestionar amb ell dades dinàmiques ens cal la utilització dels punters.

1.1. Aprofundiment en el tipus punter del llenguatge C

Anem a aprofundir en la gestió del tipus punter del llenguatge C. Recordem, abans, que ja coneixem el tipus punter a causa de la seva obligada utilització per a efectuar el pas d'arguments per referència en la crida a funcions en llenguatge C.

Enumerem els continguts que suposem ja coneguts:

- Declaració de punters.
- Assignació de valor a punter.
- Accés a una dada via punter.
- Importància del tipus d'objecte a apuntar.
- Punters en el pas d'arguments per referència.
- Error d'accés a punter `NULL`.
- Punters i taules.
- Punters a estructures.

Partint de la base que tenim assumits els coneixements corresponents als continguts anteriors, continuem amb l'estudi del tipus punter en el llenguatge C.

1.1.1. Punters genèrics

Un punter genèric és un punter de tipus `void *`, és a dir:

```
void *p;
```

Aquests punters s'utilitzen per a emmagatzemar adreces, però no es poden utilitzar per a gestionar les dades emmagatzemades en aquestes adreces (amb l'operació d'indirecció *, és clar) ja que no són de cap tipus en concret, de manera que el llenguatge C no sap com interpretar les dades apuntades.

Imaginem-nos que escrivim un codi semblant a:

```
void *p;
int x = 1000;
...
p = &x;   /* p està apuntant la direcció de memòria on hi ha x */
*p = 2000;   /* voldríem canviar el valor emmagatzemat en x a través de p */
```

El compilador es queixarà en la darrera instrucció amb un missatge semblant a `Not an allowed type`, que vol indicar que no és permès l'operador d'indirecció amb un punter `void *`.

1.1.2. Els punters i l'especificador `const`.

Recordeu que una dada constant, que es declara amb la partícula `const` (no la confongueu amb les constants simbòliques que es declaren amb la directiva de precompilació `#define`), és una dada que ocupa un espai de memòria, al qual s'accedeix a través del nom de la constant i que pren un valor, en el moment de la seva declaració, el qual es manté constant mentre la dada resideix en memòria.

L'afirmació anterior és certa a mitges. Hi manca una observació important. El valor emmagatzemat en una dada constant no és modificable a través de la dada (constant), però com que és en memòria, s'hi pot accedir a través d'un punter i, d'aquesta manera, modificar-ne el valor emmagatzemat. És a dir:

```
const int n=10;
int *p;
...
n = 20; /* error en temps de compilació */
...
p = &n;
*p = 20; /* s'ha modificat el contingut de n */
```

Per tant, podríem dir que l'especificació `const` per a una dada fa constant l'accés a la posició de memòria a través del nom de la dada.

L'especificació `const` és utilitzable, també, en la declaració de punters. Tenim dues possibles utilitzacions:

1) Punter a constant

En aquest cas el valor apuntat pel punter es comporta com a constant (no modificable) quan s'hi accedeix a través del punter. La seva sintaxi és:

```
const <tipus_dada> *<nom_punter>;
```

2) Punter constant

En aquest cas la direcció de memòria emmagatzemada en el punter és constant. La seva sintaxi és:

```
<tipus_dada> *const <nom_punter>;
```

Evidentment, les dues utilitzacions es poden combinar per a aconseguir un punter amb dues particularitats: que la direcció que emmagatzema sigui constant i que la dada apuntada per aquesta direcció es comporti

com a constant quan s'hi accedeix a través del punter. Estem davant un punter constant a constant:

```
const <tipus_dada> *const <nom_punter>;
```

Vegem-ne alguns exemples:

```
int i = 30;
const int j = 20; /* inicialització obligatòria en aquest moment */
const int *pi = &i;
int *const pj = &j; /* inicialització obligatòria en aquest moment */
const int *const pc = &i; /* inicialització obligatòria en aquest moment */
```

Imaginem que utilitzem les dades anteriors en les instruccions següents, suposant que en cada instrucció partim de la situació donada en les declaracions anteriors.

```
i = 300; /* correcte */
j = 200; /* error ja que j és constant */
pi = &j; /* correcte */
*pi = 300; /* error ja que la dada apuntada a través de pi és constant */
pj = &i; /* error ja que l'adreça que guarda pj és constant */
*pj = 200; /* correcte */
pc = &j; /* error */
*pc = 300; /* error */
```

1.1.3. Importància dels punters en la definició de taules

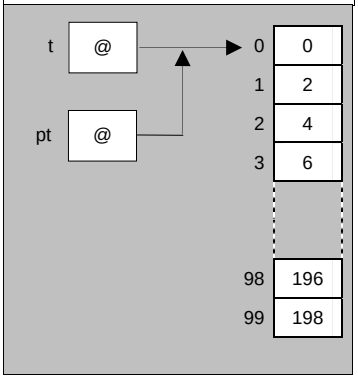
Recordem que el nom d'una taula de tipus `T` no és altra cosa que un punter a aquest tipus que apunta a la primera posició de la taula i que podem accedir a la taula amb l'operador d'indexació i amb l'operador d'indirecció.

```
unsigned int i;
int t[100];
for (i=0; i<100; i++) t[i] = 2*i;  /* equivalent a *(t+i) = 2*i; */
```

L'emplenat de la taula anterior també s'hauria pogut fer amb l'ajut d'un punter auxiliar:

```
int *pt = t;
for (i=0; i<100; i++) { *pt = 2*i; pt++; }
```

Figura 1. Distribució en memòria d'una taula.



Cal tenir molt clar, també, quin és l'espai que ocupen en memòria les declaracions dels exemples anteriors. Tal com podeu observar a la figura 1, la taula `t` genera, en realitat, un punter que conté la direcció inicial de l'espai reservat per a les 100 posicions de la taula. El punter `pt`, que hem declarat posteriorment, en haver-lo inicialitzat amb el valor de `t`,

està apuntant, també, la posició inicial de l'espai reservat per a la taula. Sembla que `t` i `pt` són equivalents. Però en realitat no ho són. El punter `t` és constant i no pot modificar el seu contingut, ja que en cas de fer-ho es perdria la direcció de la primera posició de la taula. En canvi, `pt` pot anar modificant el seu contingut, tal com s'ha vist en el darrer exemple.

D'altra banda, tenim clar que una cadena no és altra cosa que una taula de caràcters amb una marca especial de final de cadena (el zero binari). Per tant, qualsevol tractament sobre les taules genèriques es pot efectuar sobre cadenes.

Fins aquest moment, les definicions de cadenes han estat del tipus següent:

```
char cad1[100]; /* cadena de 100 caràcters */
char cad2[]="Hola"; /* cadena de 5 caràcters (zero binari!) */
```

En les dues definicions, `cad1` i `cad2` han provocat la reserva, en memòria, de l'espai corresponent al punter (de 100 i 5 octets, respectivament). És possible definir l'anterior cadena `cad2` amb la sintaxi següent:

```
char *cad2 = "Hola";
```

Aquesta definició provoca que internament s'hagin reservat l'espai corresponent al punter i, a més, 5 octets (quatre lletres i el caràcter '\0').

Cal anar molt amb compte amb declaracions d'aquest tipus: ‼

```
char *s;
```

Fixeu-vos que s'està produint la reserva de l'espai corresponent al punter, però en canvi no hi ha cap reserva d'espai per als possibles caràcters d'una cadena apuntada per `s`. Aquest punter ens podrà servir per a una assignació dinàmica de memòria en temps d'execució.

1.1.4. Taules de punters

De la mateixa manera que sabem gestionar taules dels tipus facilitats pel llenguatge C (`int`, `float`, `char`...) i de tipus definits pel programador (estructures), també podem gestionar taules de tipus punter:

```
<tipus_dada> *t[MAX]; /* on MAX és constant simbòlica */
```

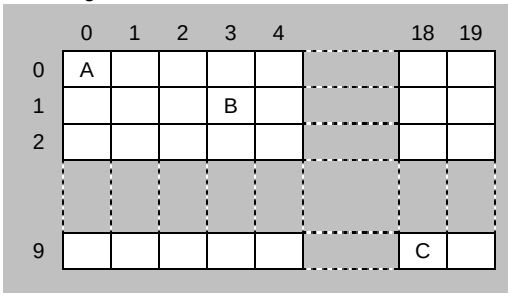
El fet de tenir una taula que emmagatzemi adreces de memòria pot ser útil en diferents situacions, sobretot quan es desitja simular amb una taula unidimensional una taula multidimensional.

El llenguatge C permet definir taules multidimensionals. Fixeu-vos en aquesta declaració:

```
int t[10][20];
```

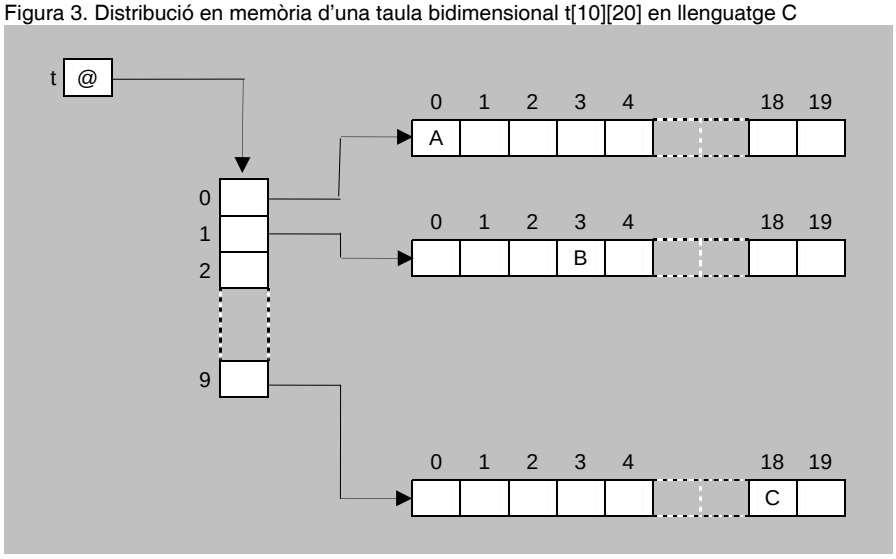
Correspon a una taula de dues dimensions, la qual hem d'interpretar com un tauler de 10 files (numerades de 0 a 9) i 20 columnes (numerades de 0 a 19), com s'observa a la figura 2. Cada casella està destinada a contenir un valor de tipus `int`.

Figura 2. Taula de 10 files i 20 columnes.



En realitat, el llenguatge C crea una taula de 10 punters a taules de 20 cel·les cadascuna, com ens mostra la figura 3.

Figura 3. Distribució en memòria d'una taula bidimensional t[10][20] en llenguatge C



Però, el llenguatge C també ens assegura que les 10 taules de 20 cel·les estan emmagatzemades en memòria consecutivament. És a dir, en l'exemple presentat, el llenguatge C provoca una reserva de 200 caselles consecutives en memòria, corresponent a la seqüència de les caselles col·locades en fila, una al costat de l'altra. És a dir:

– la casella que conté el valor A és accessible per `t[0][0]` o `*t[0];`

– la casella que conté el valor B és accessible per `t[1][3]` o `*(t[0]+1*20+3)`, ja que la casella de la fila 2 - columna 3 es troba en la posició 23 (començant a numerar per zero) si situem les files en seqüència, l'una darrere l'altra;

– la casella que conté el valor C és accessible per `t[9][18]` i també per `*(t[0] + 198),` ja que 198 és el resultat de 9*20 + 18.

En tractar-se d'una taula bidimensional, el nom de la taula (`t`) és un doble punter que apunta a una taula de punters, on cada punter apunta a la taula corresponent a cada fila. Així, `t[0]` és un punter que conté la direcció de la primera cel·la de la primera fila; `t[1]` és un punter que conté la direcció de la primera cel·la de la segona fila; i així successivament. Però, a més, resulta que aquestes files estan consecutives en memòria i, per tant, donat que cada fila és de 20 columnes, resulta que les següents igualtats (de direccions) són certes:

```
t[1] == t[0]+20
t[2] == t[1]+20 == t[0]+40
t[3] == t[2]+20 == t[1]+40 == t[0]+60
```

I, en general, `t[i]+j == t[0]+i*20+j`

Totes les expressions indicades són direccions de memòria. Si hi posem un `*` al davant es converteixen en la dada existent on indica la corresponent direcció i, per tant, les següents igualtats (de caràcters doncs la taula estava definida de caràcters), són certes:

```
*(t[0]) == t[0][0]
*(t[1]) == *(t[0]+20) == t[1][0]
*(t[2]) == *(t[1]+20) == *(t[0]+40) == t[2][0]
```

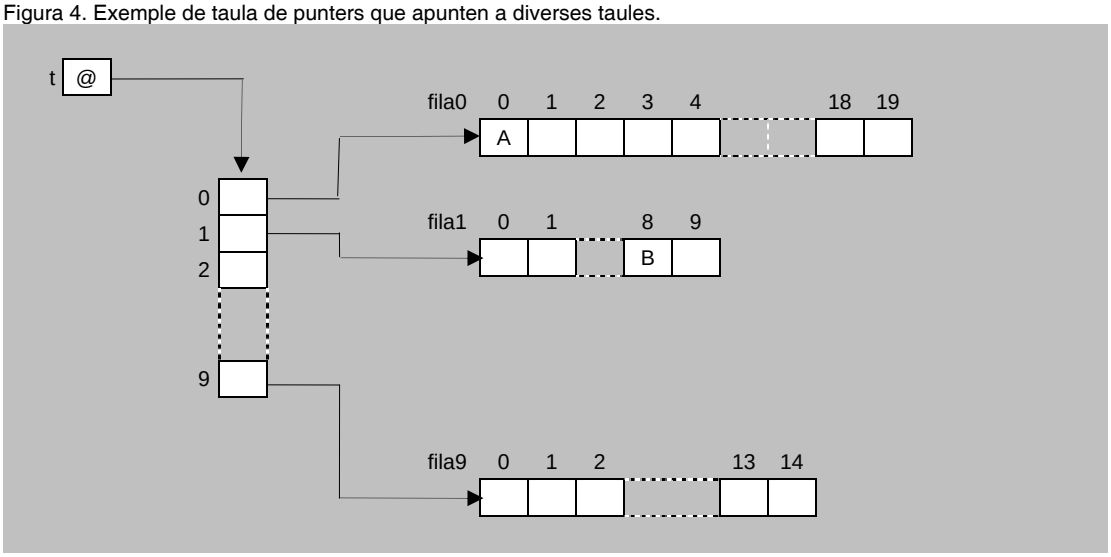
I, en general, `*(t[i]+j) == t[i][j] == *(t[0]+i*20+j)`

Ara bé, interessa tenir una estructura semblant, les files de la qual no estiguin consecutives en la memòria? O, fins i tot, volem que no totes les files tinguin la mateixa longitud? En aquest cas utilitzarem una taula de punters on cada punter contindrà la direcció d'una taula unidimensional corresponent a cada fila. Així, per exemple:

```
int fila0[20], fila1[10], ..., fila9[15];
int *t[10];
t[0]=fila0; t[1]=fila1; ..., t[9]=fila9;
```

D'aquesta manera aconseguim una situació com la representada en la figura 4.

Figura 4. Exemple de taula de punters que apunten a diverses taules.



Observem que:

Doble punter

Un doble punter a un tipus T no és res més que un punter destinat a apuntar un punter destinat a apuntar una dada de tipus T.

- la casella que conté el valor A és accessible per `fila0[0]` o `t[0][0]` o `(*t)[0]` o `**t;` (doble punter!)

- la casella que conté el valor B és accessible per `fila1[8]` o `t[1][8]` o `(*(t + 1))[8]` o `*(*(t + 1) + 8);`

- la casella que conté el valor C és accessible per `fila9[18]` o `t[9][18]` o `(*(t + 9))[18]` o `*(*(t + 9) + 18).`

Totes les possibilitats anteriors són el resultat de combinar el fet que els noms de les taules siguin punters a la primera posició de memòria corresponent a la taula i el fet d'utilitzar l'aritmètica de punters. Quina és la millor utilització? Només hi ha una resposta: la que a vosaltres us sigui més planera. Això no impedeix que hàgiu de conèixer totes les possibilitats i que les sapigueu utilitzar, ja que de vegades us podeu trobar un codi desenvolupat per altres programadors, els quals han utilitzat altres tècniques.

Una darrera pregunta: té sentit utilitzar taules de punters? La resposta és afirmativa. En moltes ocasions necessitem una estructura bidimensional per a emmagatzemar valors. Suposem que en temps de codificació del programa coneixem el nombre de files, però que desconeixem el nombre de columnes, que, fins i tot, pot variar per a cada fila en temps d'execució. En aquest cas, podrem definir una taula de punters amb una dimensió equivalent al nombre de files i deixar per al moment d'execució la definició dinàmica de les columnes a apuntar per a cada fila, és a dir, acomodarem la seva dimensió a les necessitats de l'execució.

Evidentment, per a poder fer això necessitem utilitzar l'assignació dinàmica de memòria. Ben aviat veurem en què consisteix.

1.1.5. Punters a punters

En treballar amb taules bidimensionals utilitzant els punters apareixen expressions del tipus `**t` o `*(*(t + 2) + 3)`. Podem observar que estem utilitzant doblement l'operador d'indirecció per a accedir a certes dades de tipus `int` des de la variable `t`. Això vol dir que la variable `t` és doble punter al tipus `int`.

Si davant d'una declaració del tipus:

```
int t[10];
```

diem que `t` és un punter al tipus `int`, per coherència davant d'una declaració com:

```
int *x[10];
```

haurem de convenir que `x` és un punter al tipus `int *`, és a dir, un doble punter al tipus `int`.

De la mateixa manera que davant la declaració anterior de la taula `t`, podem definir que:

```
int *pt = t; /* pt conté la mateixa adreça que conté t */
```

hem d'entendre que davant la declaració de la taula `x` podem definir que:

```
int **px = x; /* px conté la mateixa adreça que conté x */
```

I també, en taules bidimensionals:

```
int mat[10][20];
int (*pmat)[20] = mat; /* pmat conté la mateixa adreça que mat */
```

de manera que podríem utilitzar `pmat` a tots els llocs on apareix `mat`. Fixem-nos que `mat` i `pmat` són punters a taules de 20 enters i, per tant, podem efectuar l'assignació `pmat = mat`. Com exemple d'aquest funcionament, vegem el següent programa, que emplena la taula `mat` utilitzant el punter `pmat` i, posteriorment en comprovem, via el nom `mat`, que ha quedat correctament emplenada.

```
/* Programa: u1n1p00.c
```

```
   Descripció: Programa per comprovar l'accés a les cel·les d'una taula estàtica
               bidimensional via un punter
   Autor: Isidre Guixà
*/

#include <stdio.h>

void main()
{
   int mat[10][20];
   int (*px)[20] = mat;
   int i,j;

   for (i=0; i<10; i++)
      for (j=0; j<20; j++)
         px[i][j]=i;

   for (i=0; i<10; i++)
   {
      for (j=0; j<20; j++) printf("%d ",mat[i][j]);
      printf("\n");
   }
}
```

Finalitzem aquesta breu introducció als punters a punters amb una pregunta semblant a la que ens fem en introduir les taules de punters: té sentit utilitzar un doble punter? La resposta també és afirmativa. Sovint necessitem una estructura bidimensional per a emmagatzemar valors. Suposem que en temps de codificació del programa ni tan sols coneixem el nombre de files, el qual pot variar segons l'execució. En aquest cas, podrem definir un doble punter i deixar per al moment d'execució la definició dinàmica de les files. Posteriorment, podrem definir, també dinàmicament, les columnes corresponents a cada fila. Evidentment, necessitem utilitzar l'assignació dinàmica de memòria.

1.1.6. Taula de punters a cadenes

És clar que en moltes ocasions ens caldrà emmagatzemar cadenes en una taula. Suposem que volem emmagatzemar els dies de la setmana. Utilitzant una taula bidimensional hauríem de fer:

```
char d[7][10]; /* Cap dia ocupa més de 10 caràcters */
strcpy(d[0],"Dilluns");
strcpy(d[1],"Dimarts");
...
strcpy(d[6],"Diumenge");
```

Hem hagut de definir una taula de 7 files per 10 columnes, quan hi ha noms que ocupen menys de 9 caràcters. Per tant, estem desaprofitant espai. La solució és que cada fila tingui tantes posicions com siguin necessàries.

El llenguatge C ens permet declarar una taula de punters a cadenes ocupant l'espai necessari i imprescindible amb una declaració com aquesta:

```
char *d[]={"Dilluns","Dimarts","Dimecres","Dijous","Divendres","Dissabte","Diumenge"};
```

En la compilació, es crea una taula `d` de set punters a caràcter, és a dir, `char *`, on cada punter conté l'adreça a una posició de memòria on trobem els continguts `Dilluns`, `Dimarts`, `Dimecres`, `Dijous`, `Divendres`, `Dissabte` i `Diumenge`. El valor del nombre de posicions de la taula es pot forçar en la codificació del programa escrivint:

```
char *d[7]={"Dilluns","Dimarts","Dimecres","Dijous","Divendres","Dissabte","Diumenge"};
```

Si el valor que defineix la grandària de la taula de punters és superior a la quantitat de valors inicials existents entre les claus, el llenguatge inicialitza convenientment les primeres posicions de la taula i deixa les altres amb el valor `NULL`. En canvi, si el valor que defineix la grandària de la taula de punters és inferior a la quantitat de valors inicials, el compilador detectarà l'error i no generarà el codi objecte.

Observem que la variable `d` és un doble punter a `char`, però les declaracions `char *d[7]` i `char **d` no són equivalents, ja que en el primer cas es genera una taula de set punters a `char` (i, evidentment, el doble punter a `char` corresponent al nom de la taula) i en el segon cas es genera únicament el doble punter a `char` sense cap altre tipus de reserva.

1.2. Assignació dinàmica de memòria

Ha arribat, finalment, el moment esperat de conèixer els mecanismes que faciliten els llenguatges, i en especial el llenguatge C, per a la assignació dinàmica de memòria.

> L'assignació dinàmica de memòria consisteix a reservar, en temps d'execució, la quantitat de memòria necessària per a emmagatzemar dades.

Observació molt important: NO tots els llenguatges de programació disposen d'aquesta opció. (!!) El nostre bon amic C sí que la té.

Abans d'entrar, però, en les particularitats de C, cal remarcar que els llenguatges que permeten l'assignació dinàmica de memòria ho fan, principalment, d'acord amb dues accions i/o funcions. Una primera que intenta assignar la memòria necessària i una segona que allibera la memòria assignada prèviament, de manera que pugui ser utilitzada en peticions posteriors.

Per a poder treballar amb assignació dinàmica de memòria és indispensable la utilització de punters. Els hem treballat detingudament en llenguatge C. En pseudocodi, fins ara, no els havíem presentat. Ho farem a continuació.

- Declaració d'una variable `p` punter a un tipus `T`; `T` pot ser un tipus predefinit del llenguatge o un tipus de dada definida pel programador:

```
var p : ^T;
```

- Assignació de valor a una variable `p` de tipus punter, a partir de l'adreça d'una variable `d` del tipus `T`, ja existent en memòria:

```
p = ^d;
```

- Accés al valor de la dada apuntada per un punter `p`:

```
p^
```

Cal comentar l'existència d'un valor especial, el valor `NULL`, per a indicar que un punter no conté cap adreça de memòria. És important, també, notar que continua sent responsabilitat del programador inicialitzar convenientment les variables de tipus punter a `NULL` perquè no continguin valors porqueria que podrien ser interpretats com a adreces de memòria incorrectes.

Les dues accions i/o funcions bàsiques que tenen tots els llenguatges que permeten assignació dinàmica de memòria són, en pseudocodi:

```
acció assignar_memòria (var p: ^T);
```

Intenta assignar l'espai de memòria necessari per a una dada de tipus `T`, l'adreça de la qual quedarà enregistrada a la variable punter `p`. Si el valor retornat a través del paràmetre `p` és `NULL`, vol dir que no s'ha pogut assignar l'espai de memòria necessari.

```
acció alliberar_memòria (var p: ^T);
```

És el procés contrari i intenta alliberar la memòria assignada per `p` perquè quedi disponible per a altres necessitats. Suposarem que en cas de tenir èxit, el valor `p` passa a ser `NULL` (per a això ha de ser un argument per referència). Evidentment és imprescindible que la memòria assignada per `p` que volem alliberar hagi estat assignada per una instrucció d'`assignar_memòria`.

És comesa del programador alliberar la memòria assignada dinàmicament. Cal anar amb compte! Si s'assigna memòria dins una acció o funció, i en sortir de l'acció o funció no es manté cap lligam amb la zona de memòria assignada, aquesta ja no podrà ser alliberada ja que no té manera d'accedir-hi. Haurem deixat una zona de memòria inutilitzada fins que es reiniciï la màquina. ‼

1.2.1. Funcions per assignació dinàmica de memòria en C

Les funcions que aporta el llenguatge C per facilitar l'assignació dinàmica de memòria són fonamentalment dues (`malloc` i `free`), tot i que n'hi ha dues més no imprescindibles (`calloc` i `realloc`).

```
#include <stdlib.h>
void *malloc (size_t n);
/* Intenta assignar un bloc de memòria de n bytes per a emmagatzemar dades de qualsevol
tipus. Si ho aconsegueix, retorna l'adreça de memòria a partir del lloc on ha assignat
l'espai com un punter a void, el qual cal assignar a la variable punter que correspongui.
En cas de no aconseguir l'espai retorna un valor NULL. */

#include <stdlib.h>
void free (void *p);
/* Intenta alliberar el bloc de memòria apuntat per l'argument p sempre que hagi estat
assignat per les funcions malloc, calloc o realloc. */
```

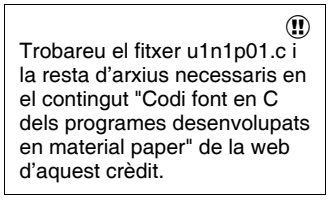
És convenient acostumar-se a alliberar la memòria assignada prèviament quan ja no sigui necessària. Un bon sistema operatiu hauria de garantir que en finalitzar l'execució d'un programa la memòria assignada dinàmicament i no alliberada fos alliberada de manera automàtica. Això, malauradament, no és així. Sistemes operatius àmpliament estesos no fan aquesta comprovació i la memòria queda amb forats fins que es reinicia la màquina.

1.2.2. Taules dinàmiques en C

En multitud d'ocasions ens apareixerà la necessitat de poder gestionar la grandària de les taules de manera dinàmica. Ara ja estem en disposició de fer-ho. I la millor manera és efectuar algun programa que ens ho permeti.

Ens interessa efectuar un programa que possibiliti a l'usuari anar introduint valors enters fins que vulgui finalitzar, de manera que pugui anar comptant els diferents valors. Deixarem que l'usuari decideixi la màxima quantitat de valors diferents a introduir.

La forma més eficient que coneixem en aquest moment per a donar solució a aquest problema consisteix en crear, una vegada l'usuari hagi indicat la màxima quantitat de valors diferents a introduir, una taula d'enters de dimensió igual al màxim nombre indicat per l'usuari. Vegem-ho.


Trobareu el fitxer u1n1p01.c i la resta d'arxius necessaris en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
/* u1n1p01.c */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "verscomp.h"

void main(void)
{
   int *p=NULL; /* punter a taula dinàmica per guardar els enters introduïts */
   unsigned int *q=NULL; /* punter a taula dinàmica per comptadors */
   unsigned int qmax; /* quantitat màxima de valors diferents */
   unsigned int qn;  /* quants enters diferents hi ha a la taula */
   int k;
   char c;

   clrscr();
   printf ("Quina quantitat màxima de valors diferents vol introduir?");
   do { k = scanf ("%u",&qmax); neteja_stdin(); } while (k == 0);
   if (qmax == 0)
   {
      printf ("No té sentit l'execució d'aquest programa.\n");
      exit (0);
   }
   p = (int *) malloc ((qmax+1) * sizeof(int)); /* Tècnica sentinella */
   if (p == NULL)
   {
      printf ("Insuficient memòria per gestionar tal quantitat de valors.\n");
      exit (1);
   }
   q = (unsigned int *) malloc (qmax * sizeof(unsigned int));
   if (q == NULL)
   {
      printf ("Insuficient memòria per gestionar tal quantitat de valors.\n");
      free (p); exit (1);
   }
   qn = 0;
   do
   {  printf ("Introdueixi un enter.\n");
      do { k = scanf ("%d",&p[qn]); neteja_stdin(); } while (k == 0);
      for (k=0; p[k] != p[qn]; k++);
      if (k < qn) q[k]++;
      else
         if (qn == qmax)
```

```
          printf ("No emmagatzemat per sobrepassar la quantitat màxima de valors
diferents.\n");
        else
          {  q[qn] = 1; qn++; }
     printf ("Vol continuar (S/N)?");
     do { k = scanf ("%c",&c); neteja_stdin(); }
     while (k == 0 || (c!='S' && c!='s' && c!='N' && c!='n'));
   } while (c=='S' || c=='s');
   if (qn == 0)
     printf ("No s'ha introduït cap valor\n");
   else
   {  clrscr();
     for (k = 0; k < qn ; k++)
       printf ("Valor %d: %u\n",p[k],q[k]);
   }
   free (p); free (q);
}
```

Observem que, tot i utilitzar punters, hem tractat les zones de memòria assignades a `p` i `q` com si fossin taules, és a dir, amb l'operador d'indexació.

Observem, també, la utilització de l'operador `sizeof` dins la crida de la funció `malloc` per a calcular el nombre de bytes necessaris. Aquesta utilització és necessària per a mantenir la compatibilitat del codi en diferents plataformes. Així, en *Turbo C*, els int ocupen 2 bytes mentre que en *gcc* n'ocupen 4. Cal utilitzar l'operador `sizeof` perquè el llenguatge en calculi el nombre que correspongui.

En l'exemple anterior hem utilitzat les funcions `malloc` i `free`. El llenguatge C incorpora una instrucció utilitzable, especialment, per a assignar memòria a una taula dinàmica. És la funció `calloc`.

```
#include <stdlib.h>
void *calloc (size_t qelem, size_t n);
/* Intenta assignar un bloc de memòria per a qelem elements de n bytes cadascun, per a
emmagatzemar dades de qualsevol tipus. Si ho aconsegueix, retorna l'adreça de memòria a
partir del lloc on ha assignat l'espai com un punter a void, que cal assignar a la
variable punter que correspongui. En cas de no aconseguir l'espai retorna un valor NULL.
*/
```

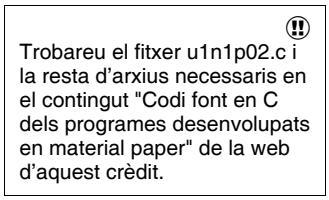
Amb la utilització d'aquesta funció, les crides a la funció `malloc` que apareixien a l'exemple anterior:

```
malloc ((qmax+1) * sizeof(int));
malloc (qmax * sizeof(unsigned int));
```

es podrien substituir per:

```
calloc (qmax+1, sizeof(int));
calloc (qmax, sizeof(unsigned int));
```

Anem a veure un nou exemple de gestió de taules dinàmiques. Es tracta d'un cas similar a l'exemple anterior, però en aquesta ocasió es tracta de comptabilitzar paraules i, per a fer una eficient gestió de la memòria, interessa emmagatzemar les paraules en l'espai adequat. Per tant, en aquest cas, hem de definir la taula que enregistra les paraules com a taula de punters a `char` de manera que a mida que l'usuari vagi introduint les paraules anirem sol·licitant memòria al sistema operatiu per tal de crear, de forma dinàmica, les cadenes on emmagatzemar les paraules.


Trobareu el fitxer u1n1p02.c i la resta d'arxius necessaris en el contingut "Codi font en C dels programes desenvolupats en material paper" de la web d'aquest crèdit.

```
/* u1n1p02.c */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include "verscomp.h"

void main()
{
   char **p=NULL; /* punter a taula dinàmica per guardar les paraules introduïdes */
   unsigned int *q=NULL; /* punter a taula dinàmica per comptadors */
   unsigned int qmax; /* quantitat màxima de paraules diferents */
   unsigned int qn;  /* quantes paraules diferents hi ha a la taula */
   int k;
   char s[255]; /* cadena reservada per a llegir les paraules */
   char c;

   clrscr();
   printf ("Quina quantitat màxima de paraules diferents vol introduir?");
   do { k = scanf ("%u",&qmax); neteja_stdin(); } while (k == 0);
   if (qmax == 0)
   {
      printf ("No té sentit l'execució d'aquest programa.\n");
      exit (0);
   }
   p = (char **) calloc (qmax, sizeof(char *)); /* Tècnica sentinella */
   if (p == NULL)
   {
      printf ("Insuficient memòria per gestionar tal quantitat de paraules.\n");
      exit (1);
   }
   q = (unsigned int *) calloc (qmax, sizeof(unsigned int));
   if (q == NULL)
   {
      printf ("Insuficient memòria per gestionar tal quantitat de paraules.\n");
      free (p); exit (1);
   }
   qn = 0;
   do
   {  printf ("Introdueixi una paraula.\n");
      do { k = scanf ("%[^,;.:?!\"\n]",s); neteja_stdin(); } while (k == 0);
      for (k=0; k<qn && strcmp (p[k],s)!=0; k++);
      if (k < qn)
         q[k]++; /* La paraula ja estava enregistrada */
      else
```

```
        if (qn == qmax)
           printf ("No emmagatzemat per sobrepassar la quantitat màxima de paraules
diferents.\n");
        else
        {
           p[qn] = (char *) calloc (strlen(s)+1, sizeof (char));
           if (p[qn] == NULL)
              printf ("Insuficient memòria per a introduir la paraula.\n");
           else
           {
              strcpy(p[qn],s);  /* s'ha emmagatzemat la paraula */
              q[qn] = 1; qn++;
           }
        }
     printf ("Vol continuar (S/N)?");
     do { k = scanf ("%c",&c); neteja_stdin(); }
     while (k == 0 || (c!='S' && c!='s' && c!='N' && c!='n'));
   } while (c=='S' || c=='s');
   if (qn == 0)
      printf ("No s'ha introduït cap valor\n");
   else
   {  clrscr();
      for (k = 0; k < qn ; k++)
      {
         printf ("Valor %s: %u\n",p[k],q[k]);
         free(p[k]);
      }
   }
   free (p); free (q);
}
```

Hi ha algunes observacions importants a fer a partir de l'exemple que acabem de veure:

- Per a efectuar la lectura de la paraula per teclat, es necessita una variable de tipus cadena i aquesta ha d'haver estat declarada com a estàtica. És la cadena `s`.

- A diferència de l'exemple anterior, no hem emprat la tècnica de sentinella, doncs per utilitzar-la necessitem definir una zona dinàmica per a guardar la paraula que acaba d'introduir l'usuari i, aquesta sol·licitud de memòria al sistema operatiu, pot resultar estèril si la paraula ja era incorporada a la taula, doncs llavors hauríem de procedir a alliberar la memòria sol·licitada.

- En l'assignació d'espai per a emmagatzemar la paraula amb el punter `p[qn]`, la quantitat de caràcters necessaris és `strlen(s) + 1`, ja que hem de pensar en el caràcter nul.

- En finalitzar el programa, abans d'alliberar l'espai assignat a través dels punters `p` i `q`, cal alliberar l'espai assignat a través de cada punter de la taula apuntada per `p`, és a dir, l'espai assignat a través de `p[0], p[1], p[2]...`

- L'assignació d'espai per la taula de punters a cadenes a través del punter `p` mereix una atenció especial:

```
p = (char **) calloc (qmax+1, sizeof (char *));
```

  La importància radica en el fet que, com que `p` és un doble punter a `char`, les posicions que es generen en aquesta assignació estan destinades a apuntar cadenes, és a dir, a ser punters a `char`, amb la qual cosa la grandària en bytes necessària per a cada una d'aquestes posicions és la grandària que necessita un punter a `char`. D'aquí el fet d'utilitzar `sizeof (char *)` per saber l'espai que ocupa un punter a `char`.

1.2.3. Reassignació de blocs de memòria en C

Sabem que les funcions `malloc` i `calloc` permeten sol·licitar al sistema operatiu, en temps d'execució, espai per a un conjunt de dades d'un determinat tipus consecutives en la memòria o, dit d'altra forma, espai per a una taula. Una vegada el sistema operatiu assigna l'espai de memòria necessari, aquest es manté inamovible al llarg de l'execució del programa fins que el propi programa n'efectua l'alliberament. I si l'espai que s'ha previst i s'ha sol·licitat esdevé massa gran o massa petit segons l'evolució de l'execució del programa? Estaria molt bé poder modificar l'espai dinàmic ja assignat en funció de les necessitats del programa. (!!)

Suposem que el punter `p` a un tipus `T` apunta a un espai dinàmic assignat pel sistema operatiu. Disposem de dos mètodes per a modificar la grandària d'aquest espai apuntat per `p`: mètode manual i mètode automàtic.

1) Mètode manual

Consisteix en sol·licitar al sistema operatiu l'assignació dinàmica d'un nou espai via un punter auxiliar `paux` amb la grandària adequada, on hi copiem les dades apuntades per `p`, en cas que el sistema operatiu ens faciliti el nou espai. En tal cas, posteriorment a la còpia cal alliberar l'espai assignat a `p` i fer una simple assignació `p = paux`, de manera que `p` apunti a la nova zona de grandària adequada.

Aquest mètode té un problema: durant un cert interval de temps d'execució, el sistema ha de tenir suficient memòria per a mantenir l'espai amb les dades originals i l'espai on copiar aquestes dades. Podria donar-se el cas que el sistema no disposés de memòria suficient per a poder efectuar el traspàs.

2) Mètode automàtic

Consisteix a utilitzar la funció `realloc` que incorpora el llenguatge C.

```
#include <stdlib.h>
void *realloc (void *p, size_t n);
/* El paràmetre p és un punter que apunta un bloc de memòria pel qual volem canviar la
quantitat de memòria assignada. Si p és NULL, aquesta funció es comporta exactament igual
que la funció malloc. Si p no és NULL i el bloc de memòria apuntat actualment ha estat
assignat amb les funcions malloc, calloc o la mateixa realloc, aquesta funció intenta
modificar la seva grandària actual i adequar-la al valor n. Les dades emmagatzemades no
canvien en el tros de bloc conservat. Aquesta funció retorna un punter al nou espai
assignat, el qual pot residir en un altre lloc de la memòria. En cas que no hi hagi prou
espai en la memòria o si n és zero, la funció retorna el valor NULL. En aquest cas, el
bloc original és alliberat. */
```

Aquest mètode té un problema: si el sistema no pot efectuar el traspàs,
hem perdut la zona de memòria assignada! Molt perillós, no us sembla?

2. Llistes

Iniciem l'estudi dels tipus de dades dinàmiques lineals que cal conèixer, ja que són utilitzables en diversos àmbits. El tipus de dada dinàmica lineal per excel·lència és la llista, objectiu d'aquest apartat. Hi ha dos tipus de dades dinàmiques lineals més, les piles i les cues, que no deixen de ser tipus especials de llista, però que per la seva importància acostumem a tenir capítols específics en tots els manuals de programació.

Llistes i taules

Les llistes, en aquest crèdit, tenen un tractament encadenat i dinàmic. Hi ha autors i llibres que presenten el concepte de llista com una taula en la qual es pot accedir al primer element i, a partir d'aquest element, accedir als següents. En aquest cas no deixem d'estar en un tractament seqüencial d'una taula, tractament que coneixem en profunditat.

Per aquest motiu, ens sembla ridícul considerar aquest tipus de llista, de manera que tractarem el concepte de llista en l'àmbit més estès, com un conjunt seqüencial d'informació encadenada amb tractament dinàmic.

Una llista simplement encadenada (anomenada llista la majoria de les vegades) és una seqüència d'elements, anomenats generalment nodes, enllaçats mitjançant punters continguts en els mateixos nodes, punters que apunten altres nodes.

Una llista s'identifica per un punter al primer node de la seqüència.

En particular, una llista pot ser buida (no hi ha seqüència d'elements) i es reconeix per què el seu punter al primer node conté el valor `NULL`.

En general, un node consta de dues parts:

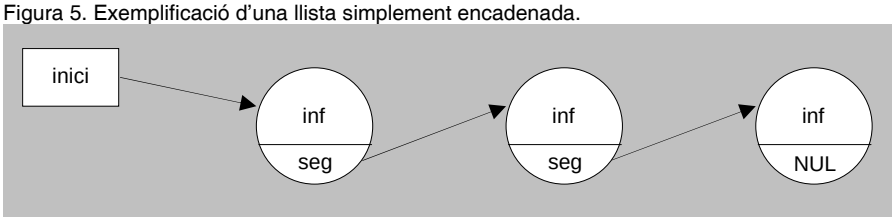
- Una part que conté la informació que es vol emmagatzemar a la llista.
- Una part que conté el lligam cap a un altre node de la llista. El darrer node de la llista conté el valor NUL com a lligam.

Per tant, si volem gestionar una llista simplement encadenada de dades d'un cert tipus `T`, caldrà tenir en compte la següent definició de node:

```
tipus node = tupla
               inf : T; /* conté la informació */
               seg : ^node; /* punter a següent node */
             fitupla
```

La figura 5 ens mostra una llista simplement encadenada de tipus `T`.

Figura 5. Exemplificació d'una llista simplement encadenada.



Bé, ja sabem com és una llista simplement encadenada. De la figura 5 se'n dedueix que per anar a un node no hi ha més remei que efectuar un recorregut seqüencial des del punter `inici` i que, una vegada estem en un node, només es pot avançar cap als nodes posteriors però no es pot retrocedir als nodes anteriors. Hi ha altres tipus de llistes amb millors prestacions però més complicades de gestionar. Centrem-nos, primer, en els algorismes de gestió de llistes simplement encadenades i amb l'experiència adquirida ja podrem intentar la incursió en la gestió de llistes més complicades.

2.1. Llistes simplement encadenades

Analitzem, a continuació, els algorismes de tractament de llistes simplement encadenades que cal tenir sempre ben presents.

1) Crear una llista buida

```
var llista: ^node;
llista = NUL;
```

Evidentment, crear la llista només consisteix a crear el punter inicial. Cal tenir, però, la precaució d'inicialitzar-lo amb el valor `NUL`.

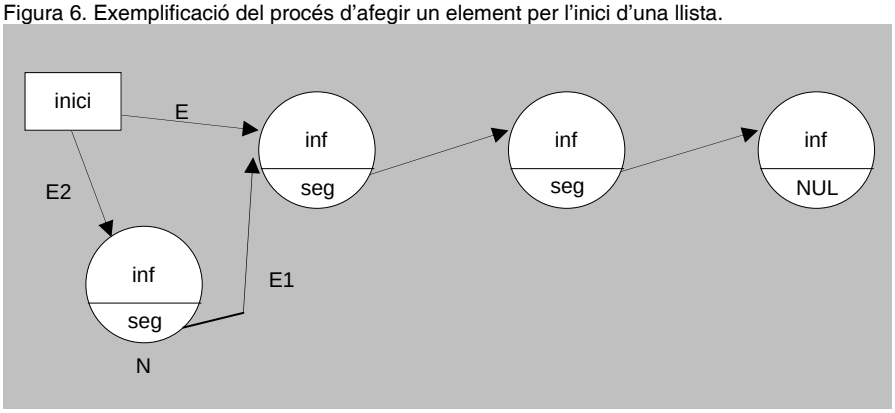
2) Afegir un element per l'inici

La figura 6 ens ajudarà a fer el seguiment del procés d'afegir un element per l'inici de la llista.

El procés d'afegir per l'inici consisteix a crear un nou node `N`, omplir-lo amb la informació que es vol afegir a la llista i afegir-lo a la llista, de manera que el punter inicial, que en un principi està apuntant el node `A`

(fletxa E), passi a apuntar el nou node N (fletxa E2), i aquest passi a apuntar el node A (fletxa E1).

Figura 6. Exemplificació del procés d'afegir un element per l'inici d'una llista.



L'algorisme en pseudocodi seria:

```
funció afegirInici (var inici:^node, element : T) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Dada a afegir a la llista
   Retorna:
      0 : Operació correcta
      1 : Error per manca de memòria
*/
   var aux:^node; fivar
   assignar_memòria (aux);
   si aux == NUL llavors retorna 1; fisi
   aux^.inf = element; aux^.seg = inici; inici = aux;
   retorna 0;
fifunció
```

Per a poder efectuar la creació del node N, ha calgut la utilització d'una variable aux punter a tipus T. No podem fer aquesta tasca amb el punter inici ja que perdria la informació que conté, i que és la que ens adreça al node A.

Observeu, també, que l'argument inici que es passa a la funció afegirInici està declarat com a pas per referència. Això és degut al fet que la funció afegirInici ha de modificar el valor que contenia el punter inici. Noteu, també, que aquesta funció té un bon funcionament si la llista és buida, ja que afegirInici, en aquest cas, procedeix a inserir el primer element a la llista.

Finalment, cal deixar constància que l'ordre en què s'efectuen les assignacions és fonamental. Una vegada el nou node està creat i tenim la seva adreça en el punter auxiliar aux, omplim el camp contenidor d'informació amb l'element que s'ha passat per valor i. El segon pas és omplir el seu punter al següent node, cosa que s'aconsegueix copiant a aux^.seg el contingut d'inici (hem assolit la fletxa E1). El tercer pas és omplir inici amb l'adreça del nou node, la qual tenim a aux (hem

assolit la fletxa E2). Si executéssim el tercer pas abans del segon, en efectuar el segon ja no disposaríem a `inici` del valor que ens porta al node A.

3) Recórrer la llista

Ja sabem que el recorregut en una llista és sempre seqüencial i, en les llistes simplement enllaçades, només es pot anar cap endavant.

Observem que per a poder recórrer la llista cal la utilització d'un punter auxiliar, ja que si utilitzéssim el punter `inici` perdríem l'adreça on es troba el primer node. Fixeu-vos, també, que el recorregut finalitza quan es troba un node que té el valor `NUL` en el seu apartat `seg`. El recorregut ja no comença si el punter `inici` té el valor `NUL` (llista buida).

L'algorisme en pseudocodi seria:

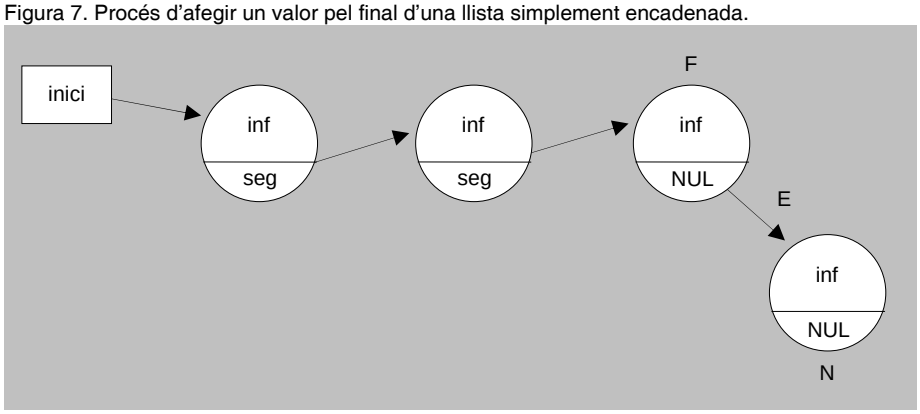
```
/* Suposem que inici:^node és la variable punter que apunta el
primer node de la llista */

var aux:^node; fivar
aux = inici;
mentre aux != NUL fer
   tractar_element (aux^.inf);
   aux = aux^.seg;
fimentre
```

4) Afegir un element pel final

La figura 7 ens ajudarà a fer el seguiment del procés d'afegir un element pel final de la llista.

Figura 7. Procés d'afegir un valor pel final d'una llista simplement encadenada.



Afegir un element pel final implica, necessàriament, un recorregut per la llista per tal d'arribar al darrer node i procedir a afegir el nou element.

El procés d'afegir pel final consisteix a crear un nou node N i omplir-lo amb la informació que es vol afegir a la llista, de manera que el punter del darrer node (F en el gràfic) deixi de tenir el valor NUL com a punter a següent node i passi a apuntar el nou node N.

L'algorisme en pseudocodi seria:

```
funció afegirFinal (var inici:^node, element : T) retorna
natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Dada a afegir a la llista
   Retorna:
      0 : Operació correcta
      1 : Error per manca de memòria
*/
   var aux1, aux2:^node; fivar
   assignar_memòria (aux1);
   si aux1 == NUL llavors retorna 1; fisi
   aux1^.inf = element; aux1^.seg = NUL;
   si inici == NUL llavors inici = aux1;
   sinó aux2 = inici^.seg;
        mentre aux2^.seg != NUL fer aux2 = aux2^.seg fimentre
        aux2^.seg = aux1;
   fisi
   retorna 0;
fifunció
```

Per a que aquest algorisme sigui correcte, cal la utilització de dues variables punters auxiliars. En efecte, una primera variable aux1 punter a tipus T és necessària per a crear el nou node abans de fer cap recorregut i poder detectar d'aquesta manera que no hi ha problemes deguts a la manca de memòria. Una vegada tenim el node N creat i omplert adequadament amb l'element que es vol afegir a la llista i apuntant NUL, hem de fer que el darrer element de la llista deixi de ser-ho i passi a apuntar el nou node. Però, quin és el darrer element de la llista?

Cal tenir present que la llista pot estar buida i, en aquest cas, el darrer element no existeix i només ens hem de preocupar que el punter inici passi a apuntar el nou node, cosa que aconseguim omplint-lo amb l'adreça que conté aux1. Si el punter inici no té el valor NUL, té sentit preguntar pel contingut d'inici^.seg, valor que situem en la segona variable aux2 punter a T. Aquest punter l'utilitzem per a situar-nos en el darrer node F de la llista. Una vegada allà només ens falta fer que el seu punter a següent passi a apuntar el node N, l'adreça del qual tenim en la variable aux1.

Finalment, cal fer notar que aquí també és imprescindible que l'argument `inici` es passi per referència, ja que el seu contingut serà modificat en cas que la llista sigui buida.

5) Cercar un element

Per a poder realitzar una operació d'aquest estil cal tenir un criteri de recerca. Cal prendre algunes decisions.

- Com podem detectar l'element a cercar en un recorregut per la llista? Hi ha diverses possibilitats: coincidència en el valor d'un, alguns o tots els camps de l'element. En el darrer cas seria una recerca destinada només a saber si l'element és a la llista, ja que no tindria sentit cercar-lo per saber dades si ja les coneixem totes. Com que estem plantejant el problema en una situació teòrica, considerarem que hi ha un conjunt de camps que formen un tipus `T*`, subtipus de `T`, pels quals s'ha d'efectuar la recerca. Per tant, caldrà passar com a argument un element del tipus `T*` que contingui els camps de recerca.

- Com podem actuar si hi ha la possibilitat d'elements repetits a la llista? Prenem la decisió de considerar la primera ocurrència trobada de l'element cercat.

Sota aquests supòsits, tenim que:

```
funció recerca (inici:^node; var element: T; referència: T*) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Variable retornada plena amb el valor trobat
      referència: Dada de tipus T* a cercar
   Retorna:
      0 : Recerca efectuada amb èxit.
      1 : Recerca efectuada sense èxit.
*/
   mentre inici != NUL i no coincideix (inici^.inf, referència) fer
      inici = inici^.seg;
   fimentre
   si inici == NUL llavors retorna 1;
   sinó element = inici^.inf; retorna 0;
   fisi
fifunció

funció coincideix (element: T, referència: T*) retorna booleà és
   /* Codi que comprovi si element correspon a referència */
fifunció
```

Observeu que dins la funció no utilitzem cap variable punter auxiliar per recórrer la llista ja que emprem el mateix punter `inici`. És això lícit? ¿No estarem perdent l'adreça del primer node de la llista? Evidentment

sí que la perdem dins la funció, però en finalitzar-ne l'execució el punter continua tenint el mateix valor inicial, ja que s'ha passat per valor i no per referència.

6) Inserir un element

Per a poder realitzar una operació d'aquest estil cal tenir com a referència un element de la llista respecte al qual s'insereix el nou element (abans o després). Per a poder realitzar aquest tipus de tractament considerem els mateixos supòsits que hem tingut en compte en la recerca d'un element.

```
funció inserirAbans (var inici:^node; element: T, referència: T*) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Dada a afegir a la llista
      referència: Dada de tipus T* a cercar per a inserir, abans d'ella, l'element
   Retorna:
      0 : Operació correcta
      1 : Error per manca de memòria
      2 : No es troba la referència
*/
   var pref, ant, aux:^node; fivar
   pref = inici; ant = NUL;
   mentre pref != NUL i no coincideix (pref^.inf, referència) fer
      ant = pref; pref = pref^.seg;
   fimentre
   si pref == NUL llavors retorna 2; fisi
   assignar_memòria (aux);
   si aux == NUL llavors retorna 1; fisi
   aux^.inf = element; aux^.seg = pref;
   si pref == inici
     llavors inici = aux; /* l'element referència era el primer de la llista */
     sinó ant^.seg = aux; /* l'element referència no era el primer de la llista */
   fisi
   retorna 0;
fifunció
```

Observem que l'argument `inici` cal passar-lo per referència, ja que pot quedar modificat si la dada referenciada és la que estava en primer lloc.

```
funció inserirDesprés (inici:^node; element: T, referència: T*) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Dada a afegir a la llista
      referència: Dada de tipus T* a cercar per a inserir, després d'ella, l'element
   Retorna:
      0 : Operació correcta
      1 : Error per manca de memòria
      2 : No es troba la referència
*/
   var pref, aux:^node; fivar
   pref = inici;
   mentre pref != NUL i no coincideix (pref^.inf, referència) fer
```

```
      pref = pref^.seg;
   fimentre
   si pref == NUL llavors retorna 2; fisi
   assignar_memòria (aux);
   si aux == NUL llavors retorna 1; fisi
   aux^.inf = element; aux.seg = pref^.seg; pref^.seg = aux;
   retorna 0;
fifunció
```

Observem que l'argument `inici` no cal passar-lo per referència, ja que mai no pot resultar modificat per un `inserirDesprés`.

7) Esborrar el primer element

Quan es parla d'esborrar un element d'una llista (sigui el primer, el darrer o un entremig), podem considerar la funció que només esborra o la funció que esborra i retorna l'element eliminat. De fet, aquesta segona opció és molt més interessant i no suposa cap cost de programació. En les operacions d'esborrament que presentem considerarem un argument per a recuperar l'element esborrat.

```
funció esborrarPrimer (var inici:^node, var element: T) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Variable on s'ha de recollir l'element esborrat
   Retorna:
      0 : Operació correcta
      1 : Error ja que la llista és buida
*/
   var aux:^node; fivar
   si inici == NUL llavors retorna 1; fisi
   aux = inici^.seg; element = inici^.inf; alliberar_memòria (inici);
   inici = aux; retorna 0;
fifunció
```

8) Esborrar el darrer element

```
funció esborrar_darrer (var inici:^node, var element: T) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Variable on s'ha de recollir l'element esborrat
   Retorna:
      0 : Operació correcta
      1 : Error ja que la llista és buida
*/
   var ult, ant: ^node; fivar
   si inici == NUL llavors retorna 1; fisi
   ult = inici; ant = NUL;
   mentre ult^.seg != NUL fer ant = ult; ult = ult^.seg; fimentre
   si ult == inici llavors inici = NUL; sinó ant^.seg = NUL; fisi
   element = ult^.inf; alliberar_memòria (ult); retorna 0;
fifunció
```

9) Esborrar un element

Ens trobem en la mateixa situació que en l'algorisme corresponent a inserir un element. Cal tenir una referència per a saber quin element (un en concret, el d'abans o el de després) s'ha d'esborrar. Tindrem en compte els mateixos supòsits d'aquella situació.

```
funció esborrarElement (var inici:^node; var element: T, referència: T*) retorna natural
és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Variable on s'ha de recollir l'element esborrat
      referència: Dada de tipus T* corresponent a l'element a esborrar
   Retorna:
      0 : Operació correcta
      1 : No es troba la referència
*/
   var pref, ant: ^node; fivar
   pref = inici; ant = NUL;
   mentre pref != NUL i no coincideix (pref^.inf, referència) fer
      ant = pref; pref = pref^.seg;
   fimentre
   si pref == NUL llavors retorna 1; fisi
   si pref == inici
     llavors inici = pref^.seg; /* l'element referència era el primer de la llista */
     sinó ant^.seg = pref^.seg; /* l'element referència no era el primer de la llista */
   fisi
   element = pref^.inf; alliberar_memòria (pref); retorna 0;
fifunció

funció esborrarAnterior (var inici:^node; var element: T, referència: T*) retorna
       natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Variable on s'ha de recollir l'element esborrat
      referència: Dada de tipus T* a cercar per a esborrar l'element anterior
   Retorna:
      0 : Operació correcta
      1 : No es troba la referència
      2 : No existeix un element anterior a la referència
*/
   var pref, ant1, ant2: ^node; fivar
   pref = inici; ant1 = NUL; ant2 = NUL;
   mentre pref != NUL i no coincideix (pref^.inf, referència) fer
      ant2 = ant1; ant1 = pref; pref = pref^.seg;
   fimentre
   si pref == NUL llavors retorna 1; fisi
   si ant1 == NUL llavors retorna 2; fisi
   si ant1 == inici
     llavors inici = pref; /* l'element a esborrar era el primer de la llista */
     sinó ant2^.seg = pref; /* l'element a esborrar no era el primer de la llista */
   fisi
   element = ant1^.inf; alliberar_memòria (ant1); retorna 0;
fifunció

funció esborrarSegüent (inici:^node; var element: T, referència: T*) retorna natural és
/* Arguments:
```

```
      inici: Punter que apunta l'inici de la llista
      element: Variable on s'ha de recollir l'element esborrat
      referència: Dada de tipus T* a cercar per a esborrar l'element posterior
   Retorna:
      0 : Operació correcta
      1 : No es troba la referència
      2 : No existeix un element posterior a la referència
*/
   var pref, aux:^node; fivar
   pref = inici;
   mentre pref != NUL i no coincideix (pref^.inf, referència) fer
      pref = pref^.seg;
   fimentre
   si pref == NUL llavors retorna 1; fisi
   si pref^.seg == NUL llavors retorna 2; fisi
   aux = pref^.seg; pref^.seg = aux^.seg;
   element = aux^.inf; alliberar_memòria (aux); retorna 0;
fifunció
```

Observem que l'argument `inici` no cal passar-lo per referència, ja que mai no pot resultar modificat per un `esborrarSegüent`.

10) Eliminar la llista

És molt important tenir un algorisme que ens elimini tota una llista. Aquest algorisme l'executarem quan ja no ens interessi tenir la llista dins una execució i, sobretot, en finalitzar l'execució d'un programa per tal d'eliminar totes les llistes creades dinàmicament.

L'algorisme és, evidentment, molt senzill. Simplement cal fer un recorregut per tots els nodes i alliberar la memòria assignada corresponent.

```
acció eliminar_llista (var inici:^node) és
/* Argument inici: Punter que apunta a l'inici de la llista */
   var aux: ^node; fivar
   mentre inici != NUL fer
      aux = inici^.seg; alliberar_memòria (inici); inici = aux;
   fimentre
fiaccióó
```

Fixeu-vos que en sortir de l'acció, el punter `inici` té el valor `NUL`, com és lògic.

2.2. Llistes simplement encadenades ordenades sense repetits

Coneixem, doncs, els algorismes de gestió d'una llista simplement encadenada genèrica en la qual no hi ha establert cap tipus d'ordre entre els seus elements ni tampoc cap tipus d'identificació unívoca, és a dir, hi pot haver elements repetits.

¿Tenen sentit tots els algorismes sobre llistes simplement encadenades en cas que la llista es mantingui ordenada sota algun criteri? Evidentment no, ja que no té sentit afegir per l'inici ni pel final, ni abans ni després d'un cert element. Si es tracta d'afegir ordenadament, només hi ha una possibilitat: el lloc que li correspon segons l'ordre establert. De manera semblant es pot raonar el sentit dels algorismes de recerca i d'eliminació.

Anem a estudiar els algorismes d'inserció, recerca i eliminació, suposant que no hi pot haver elements repetits en el camp o conjunt de camps que formen el criteri d'ordenació. És evident que amb alguna modificació es poden dissenyar els algorismes per a gestionar llistes simplement encadenades ordenades amb elements repetits.

1) Cercar un element

Considerarem que tenim un tipus `T*` que conté els camps que identifiquen els elements i pels quals està definit l'ordre establert a la llista.

```
función recerca (inici:^node; var element: T; referència: T*) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Variable retornada plena amb el valor trobat, si existeix
      referència: Dada de tipus T* corresponent a l'element a cercar
   Retorna:
      0 : Recerca efectuada amb èxit.
      1 : Recerca efectuada sense èxit.
*/
   mentre inici != NUL i és_menor (inici^.inf, referència) fer
      inici = inici^.seg;
   fimentre
   si inici == NUL o és_major (inici^.inf, referència) llavors retorna 1; fisi
   element = inici^.inf; retorna 0;
fifunció

función és_menor (element: T, referència: T*) retorna booleà és
   /* Codi que comprovi si la part T* d'element és més petita que referència */
fifunció

función és_major (element: T, referència: T*) retorna booleà és
   /* Codi que comprovi si la part T* d'element és més gran que referència */
fifunció
```

2) Inserir un element

```
función inserir (var inici:^node; element: T) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista ordenada
      element: Dada a inserir a la llista
   Retorna:
```

```
      0 : Operació correcta
      1 : Error per manca de memòria
      2 : Ja existeix un element amb el mateix identificador
*/
   var seg, ant, aux:^node; fivar
   seg = inici; ant = NUL;
   mentre seg != NUL i no és_menor* (seg^.inf, element) fer
      ant = seg; seg = seg^.seg;
   fimentre
   si seg == NUL o és_major* (seg^.inf, element) llavors
      assignar_memòria (aux);
      si aux == NUL llavors retorna 1; fisi
      aux^.inf = element; aux^.seg = seg;
      si ant == NUL
         llavors inici = aux; /* l'element s'insereix a l'inici de la llista */
         sinó ant^.seg = aux;
      fisi
      retorna 0;
   sinó retorna 2;
   fisi
fifunció

funció és_menor* (ele1: T, ele2: T) retorna booleà és
   /* Codi que comprovi si la part T* d'ele1 és més petita que la d'ele2 */
fifunció

funció és_major* (ele1: T, ele2: T) retorna booleà és
   /* Codi que comprovi si la part T* d'ele1 és més gran que la d'ele2 */
fifunció
```

3) Esborrar un element

```
funció esborrar (var inici:^node; var element: T; referència: T*) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la llista
      element: Variable retornada plena amb l'element a esborrar, si existeix
      referència: Dada de tipus T* corresponent a l'element a esborrar
   Retorna:
      0 : Esborrament efectuat amb èxit.
      1 : Esborrament no efectuat perquè no s'ha trobat l'element.
*/
   var pref, ant: ^node; fivar
   pref = inici; ant = NUL;
   mentre pref != NUL i és_més petit (pref^.inf, referència) fer
      ant = pref; pref = pref^.seg;
   fimentre
   si pref == NUL o és_més gran (pref^.inf, referència) llavors retorna 1; fisi
   si pref == inici
      llavors inici = pref^.seg; /* l'element esborrat és l'inicial */
      sinó ant^.seg = pref^. seg
   fisi
   element = pref^.inf; alliberar_memòria (pref);
   retorna 0;
fifunció
```

2.3. Llistes circulars o en anell

Les llistes simplement encadenades no permeten, a partir d'un element, accedir directament als anteriors. Per a solucionar aquest inconvenient es pot fer que el darrer element apunti el primer i, d'aquesta manera, es pot obtenir una llista circular.

Per a accedir a una llista circular farà falta, com en les llistes simplement encadenades, un punter. En aquest cas és més avantatjós substituir el punter inicial que consideràvem en les llistes simplement encadenades per un punter final que adreci al darrer element de la llista circular. D'aquesta manera, s'obté un fàcil accés a l'element inicial i a l'element final, ja que:

Suposem que `final: ^node` és el punter al darrer element de la llista. Llavors:

- Per a accedir al darrer element de la llista només cal considerar el node `final^`. És a dir, el darrer element de la llista és `final^.inf`.

- Per a accedir al primer element de la llista només cal considerar el node `(final^.seg)^`. És a dir, el primer element de la llista és `(final^.seg)^.inf`.

Les operacions en llistes circulars són similars a les operacions de les llistes simplement encadenades, excepte en allò que fa referència al primer o darrer element de la llista circular. Deixem per a vosaltres el disseny dels algorismes corresponents.

2.4. Llistes doblement encadenades

En les llistes simplement encadenades, el recorregut només es pot efectuar en un únic sentit: de l'inici cap al final. Per a solucionar aquest inconvenient i permetre un recorregut en ambdós sentits, es poden construir llistes doblement encadenades

En aquestes llistes, cada node conté dos punters, `anterior` i `següent`, que apunten, respectivament, cap enrere i cap endavant. És a dir, si per a un node `Y` hi ha un node `X` que l'apunta amb el punter `següent`, llavors el node `X` és apuntat pel punter `anterior` del node `Y`, i si un node `Y` està apuntant un node `Z` amb el punter `següent`, llavors el node `Y` està apuntat pel punter `anterior` del node `Z`.

Com que cada element conté dos punters, una llista doblement encadenada ocupa més espai en memòria que una llista simplement encadenada per a una mateixa quantitat d'informació.

Per a accedir a una llista doblement encadenada tenim dos punters, `inici` i `final`. Com que el primer element de la llista no té un element anterior, el seu punter `anterior` conté el valor `NUL`. D'igual manera, com que el darrer element de la llista no té un element següent, el seu punter `següent` conté el valor `NUL`.

Les operacions en llistes doblement encadenades són similars a les operacions de les llistes simplement encadenades, però cal tenir en compte que hi ha els dos punters per a accedir a la llista, els dos punters de cada node i que el recorregut es pot fer en tots dos sentits. Deixem per a vosaltres el disseny dels algorismes corresponents.

També es pot considerar el cas de llistes doblement encadenades circulars, de manera que el darrer node apunta el primer amb el punter `següent` i el primer node apunta el darrer amb el punter `anterior`. No necessitem, però, dos punters per a accedir a la llista. Amb un n'hi ha prou. En aquest cas, ens preguntem quina és la millor elecció com a únic punter d'accés en una llista doblement encadenada circular: ha d'apuntar el primer o el darrer element de la llista? Què en penseu?

3. Piles i cues

Passem ara a estudiar dos tipus de dades fonamentals en informàtica: les piles i les cues. En la majoria dels nostres programes les hem estat utilitzant, de manera implícita, sense ser-ne conscients.

Les piles són, com el seu nom indica, recipients on emmagatzemar dades apilades i, com en una pila de qualsevol tipus d'elements (plats, llibres,...), les úniques operacions que es poden fer són: ficar un element damunt la pila i treure l'element del damunt (darrer element ficat a la pila).

Les cues són, com el seu nom indica, recipients on emmagatzemar dades en l'ordre en que van arribant i, com en una cua qualsevol (de persones a l'entrada d'un cinema, de cotxes en una carretera,...), les úniques operacions que es poden fer són: ficar un element al final de la cua i treure l'element de l'inici de la cua (l'element que fa més temps que és a la cua).

Per a cadascun d'aquests tipus en veurem, en primer lloc, la seva definició i la seva implementació amb la utilització dels tipus de dades coneguts; posteriorment, en presentarem diferents camps d'aplicació.

3.1. Piles

Una pila és un conjunt lineal (ja que els seus components ocupen llocs successius dins el conjunt) d'elements en el qual es poden inserir o eliminar elements només per un dels dos extrems. En conseqüència, els elements d'una pila seran eliminats en ordre invers al qual es van inserir.

A causa del funcionament d'inserció i eliminació, les piles es coneixen com a estructures LIFO (last in - first out), ja que el darrer element que va ser inserit (last in) és el primer a ser eliminat (first out).

La noció de pila ens és familiar: pila de plats, de llibres... Per a afegir un plat es col·loca dalt de la pila; per a agafar-lo es pren de la part superior de la pila.

Hi ha un concepte anglès molt utilitzat en la gestió de piles: top. Aquesta paraula indica el capdamunt i, evidentment, la gestió de piles treballa sempre amb l'element top de la pila.

Amb les piles només s'hi poden efectuar dues operacions: `posar` (molt estès el terme anglès `push`) i `treure` (en anglès `pop`). ‼

Hi ha autors que permeten una tercera operació: `capdamunt`, (en anglès, `top`) pensada per a consultar el valor que hi en el `top` de la pila. De fet, els autors que no tenen en compte aquesta operació, cada vegada que volen consultar quin és l'element que hi ha en el capdamunt de la pila han de procedir a efectuar un treure, per a veure l'element, i un posar per a tornar a posar l'element a la pila. Nosaltres permetrem la utilització d'aquesta tercera operació.

Tornem-ho a repetir. Tot i que les implementacions de pila ho permetin, el concepte de tipus de dada pila porta implícit que no s'hi pugui fer altra cosa que les dues (o tres) operacions esmentades, és a dir, no es pot realitzar un recorregut per una pila, no es pot efectuar una recerca per una pila, no es pot treure l'element de sota (ja que no s'hi té accés)... ‼

3.1.1. Implementació del tipus de dada pila

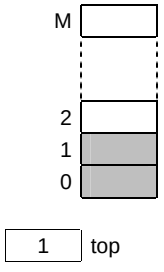
Tradicionalment hi ha dues implementacions del tipus de dada pila:

a) Mitjançant taules

En aquest cas, en la creació de la pila cal definir la grandària màxima de la pila i tenir una variable entera per a apuntar al darrer element inserit, l'element top.

La figura 8 ens mostra la implementació d'una pila mitjançant una taula de grandària M+1 que ja conté dos elements. Observem que la variable `top` apunta a la casella que conté el darrer element afegit.

Figura 8. Implementació d'una pila mitjançant una taula



Malgrat que les taules permeten accés a qualsevol element, la gestió de piles només permet accés a l'element del capdamunt, és a dir, l'element apuntat pel valor que conté top.

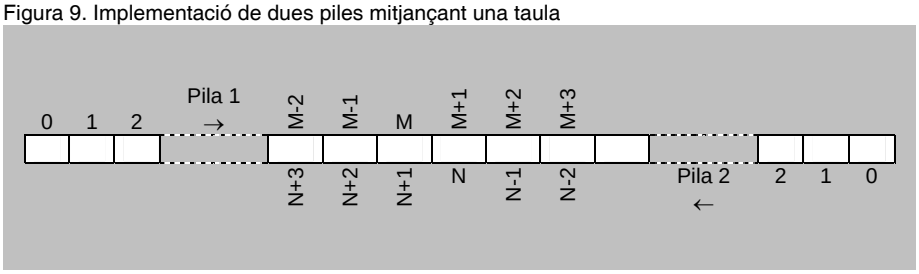
En una pila només podem afegir un nou element (el valor de top augmentaria en una unitat) o recuperar l'element del capdamunt (el valor de top disminuiria en una unitat).

La implementació de piles mitjançant taules presenta un problema: la pila té una capacitat fixa que ocupa un espai en memòria i, per tant, es desaprofita espai si la pila no és plena i no hi ha possibilitat d'augmentar la capacitat quan ja és plena.

Bé, en el cas del llenguatge C, que permet la definició de taules dinàmiques, la implementació de piles mitjançant taules no seria tan dolenta com en el cas de llenguatges que no permeten taules dinàmiques, però tot i així, sabem dels problemes que té el llenguatge C per augmentar i/o disminuir la grandària d'una taula dinàmica. I, tot i així, tindríem el problema de desaprofitament d'espai quan la pila no estès plena.

A vegades es pot intentar compartir espai de memòria per minorar, tant com es pugui, el problema presentat d'espai de memòria fix i estàtic. Aquesta gestió és possible quan cal tractar amb dues piles del mateix tipus d'element de grandàries `M+1` i `N+1`. Podem decidir gestionar-les amb una única taula de grandària `M+N+2`. La figura 9 ens mostra el muntatge.

Figura 9. Implementació de dues piles mitjançant una taula

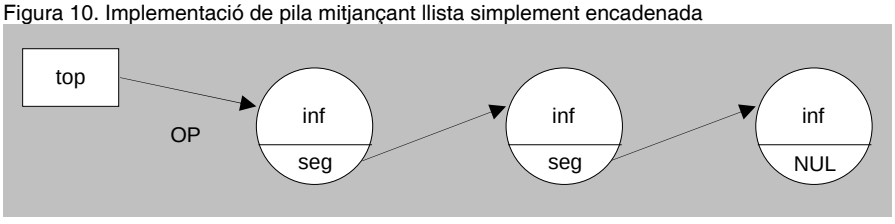


Si en algun punt del procés la `pila1` necessita més de `M+1` posicions i en aquell moment la `pila2` no té ocupades `N+1` posicions, aquesta pot cedir espai a l'altra.

b) Mitjançant llistes simplement encadenades (!!)

La figura 10 ens en mostra una exemplificació.

Figura 10. Implementació de pila mitjançant llista simplement encadenada



En aquest cas, es tracta de crear una llista simplement encadenada on l'element top és el punter a l'inici de la pila. Llavors, les dues operacions sobre piles consisteixen en dues de les operacions sobre llistes simplement encadenades:

- Posar un element a la pila (`push`) és equivalent a afegir un element per l'inici de la llista (per la zona OP de la figura 10).
- Recuperar un element de la pila (`pop`) és equivalent a esborrar un element per l'inici de la llista (per la zona OP de la figura 10) i recollint l'element eliminat de la pila via paràmetre per referència.

A més, és interessant tenir una operació per a eliminar la pila, la qual és equivalent a l'operació corresponent per a eliminar una llista.

En aquesta implementació l'única limitació es troba en la quantitat de memòria disponible. La pila pot créixer i disminuir segons les necessitats. Per tant, és la millor opció.

3.1.2. Aplicacions de les piles

Les piles s'utilitzen molt en informàtica. Enumerarem algunes de les seves aplicacions.

a) Crides a subprogrames (accions i funcions)

Quan es té un programa que crida un subprograma, internament s'utilitzen piles per a guardar l'estat de les variables del programa en el moment que es fa la crida. Així, quan s'acaba l'execució del subprograma, els valors emmagatzemats a la pila poden recuperar-se i es pot continuar l'execució del programa en el punt on va ser interromput. A més de les variables s'ha d'emmagatzemar l'adreça del programa on es va fer la crida, ja que és en aquesta posició on retorna el control del procés.

Evidentment, aquesta gestió cal fer-la amb una estructura de tipus pila, ja que l'estat de les diverses crides es va apilant en l'ordre en què es van efectuant (imagineu-vos diversos nivells de crides). A mesura que van finalitzant (ordre invers en què s'han produït), es van desapilant els corresponents estats que indiquen la situació existent abans d'efectuar les dites crides.

b) Recursivitat

La recursivitat, tema que per la seva importància es mereix un capítol específic, consisteix en la resolució d'un problema complex en termes de si mateix, però en un grau de complexitat menor, de tal manera que, a la fi, el problema queda del tot resolt.

Des del punt de vista informàtic, la recursivitat implica que un subprograma es cridi a si mateix (de manera directa o indirecta) i, per

tant, la recursivitat no deixa de ser un cas particular de crides a subprogrames.

Hi ha llenguatges de programació que no faciliten la utilització de recursivitat. En aquests casos, resulta convenient tenir algun procediment per a simular la recursivitat. La utilització del tipus de dada pila ho permetrà.

c) Tractament d'expressions aritmètiques

Un problema usual en computació és avaluar una expressió aritmètica. Com actua l'ordinador quan ha d'avaluar una operació tan simple com 10 + 3 * 5 ?

Evidentment, el resultat ha de ser 25. Quin raonament segueix l'ésser humà per a obtenir aquest resultat? En primer lloc, fent una ràpida ullada, la nostra capacitat visual ens permet captar l'expressió globalment (ja que és molt curta) i la nostra capacitat intel·lectual ens permet deduir amb rapidesa que, primer, cal efectuar el producte i, posteriorment, la suma, ja que matemàticament un producte té prioritat davant una suma.

Ara bé, ¿i si us demano que em calculeu el resultat de l'expressió 3 ^ 2 * log 25 – 43 div 4 / 25 + 2 * 43 mod 21? Em sembla que no sereu tan ràpids com abans i efectuareu una avaluació acurada de la prioritat dels operadors.

L'ordinador actua sempre de la mateixa manera; tant li fa si l'expressió és curta o si és llarga. Per a resoldre l'avaluació es pot treballar amb dues piles, una primera en la qual s'emmagatzemen els valors dels operands i de les avaluacions parcials i una segona en la qual s'emmagatzemen els operadors pendents d'execució.

Fem-ne el seguiment amb l'expressió 10 + 2 – 3 * 5 + 4. Considerarem `Val (...)` la pila que conté els valors i `Ope (...)` la pila que conté els operadors. Partim de la situació inicial en què les dues piles són buides: `Val ()` i `Ope ()`.

– Recepció del valor `10`. No es pot fer altra cosa que apilar-lo a la pila de valors.

```
Val (10)                    Ope ()
```

– Recepció de l'operador +. No es pot fer altra cosa que apilar-lo a la pila d'operadors i esperar el valor següent.

```
Val (10)                    Ope (+)
```

– Recepció del valor 2. No es pot fer altra cosa que apilar-lo a la pila de valors i esperar l'operador següent, ja que si és més prioritari que l'operador que hi ha actualment en el `top` de la pila `Ope` no es pot operar amb el valor que hi ha actualment en el `top` de la pila `Val`.

```
Val (10,2)                  Ope (+)
```

– Recepció de l'operador -. Com que el `top` de la pila `Ope` té un símbol +, que té la mateixa prioritat que el símbol que està arribant, estem en condicions d'executar l'operació pendent + amb els valors existents a la pila `Val`. Per aquest motiu desapilem dos valors de la pila `Val` (2 i 10) i un operador de la pila Ope (+) i fem l'operació corresponent (10 + 2), emmagatzemant el resultat (12) a la pila Val. Si analitzem el top de la pila Ope observem que no hi ha cap execució pendent possible. El nou operador - cal apilar-lo a la pila de valors i esperar el valor següent.

```
Val (12)                    Ope (-)
```

– Recepció del valor 3. No es pot fer altra cosa que apilar-lo a la pila de valors i esperar l'operador següent, ja que si és més prioritari que l'operador que hi ha actualment en el `top` de la pila `Ope` no es pot operar amb el valor que hi ha actualment en el `top` de la pila `Val`.

```
Val (12,3)                  Ope (-)
```

– Recepció de l'operador *. Com que el `top` de la pila `Ope` té un símbol –, que té menys prioritat que el símbol que està arribant, s'ha d'emmagatzemar l'operador entrant i executar-lo quan es pugui, però sempre abans que l'actual `top` de la pila `Ope`.

```
Val (12,3)                  Ope (-,*)
```

– Recepció del valor 5. No es pot fer altra cosa que apilar-lo a la pila de valors i esperar l'operador següent, ja que si és més prioritari que l'operador que hi ha actualment en el top de la pila `Ope` no es pot operar amb el valor que hi ha actualment en el `top` de la pila `Val`.

```
Val (12,3,5)                Ope (-,*)
```

– Recepció de l'operador +. Com que el `top` de la pila `Ope` té un símbol *, que té més prioritat que el símbol que està arribant, estem en condicions d'executar l'operació pendent * amb els valors existents a la pila `Val`. Per aquest motiu, desapilem dos valors de la pila `Val` (5 i

3) i un operador de la pila `Ope` (*), fem l'operació corresponent (3 * 5) i emmagatzemem el resultat (15) a la pila `Val`. Ens queda aquesta situació:

```
Val (12,15)                    Ope (-)
```

Encara ara, el `top` de la pila `Ope` té un símbol – que té la mateixa prioritat que el símbol + que està arribant i, per tant, estem en condicions d'executar l'operació pendent – amb els valors existents a la pila `Val`. Per aquest motiu, desapilem dos valors de la pila `Val` (15 i 12) i un operador de la pila `Ope` (–), fem l'operació corresponent (12 – 15) i emmagatzemem el resultat (–3) a la pila `Val`. Si analitzem el `top` de la pila `Ope` observem que no hi ha cap possible execució pendent. El nou operador + cal apilar-lo a la pila de valors i esperar el valor següent.

```
Val (-3)                       Ope (+)
```

– Recepció del valor 4. Com que detectem que és el darrer element de l'expressió, cal operar-lo a través de l'operador existent en el `top` de la pila `Ope` amb el valor existent en el `top` de la pila `Val`. Per això desapilem l'operador (+) de la pila `Ope` i l'operand (–3) de la pila `Val`, efectuem la darrera operació (–3 + 4) i obtenim el resultat (1) final de l'expressió inicial.

```
Val ()                         Ope ()
```

Evidentment, abans d'efectuar l'avaluació en temps d'execució, el compilador haurà efectuat una anàlisi sintàctica de l'expressió en temps de compilació per garantir que en temps d'execució no hi haurà problemes sintàctics. El procés d'avaluació del compilador també utilitza piles.

En el procés anterior comentàvem, a cada pas, que miràvem el valor del `top` de les piles. Això només ho podrem fer si disposem de l'operació `capdemunt`. En cas que no es faciliti aquesta operació, no té sentit executar les operacions apilar i desapilar per a consultar el valor del `top`. En aquest cas, podríem mantenir en altres variables els valors dels tops de les piles i desapilar només quan fos estrictament necessari per a efectuar els càlculs corresponents.

d) Algorisme d'ordenació ràpida

El mètode d'ordenació ràpida, conegut com a quicksort, és un mètode fàcilment programable utilitzant recursivitat. Per tant, es pot tenir una versió no recursiva amb la utilització de piles.

3.2. Cues

Una cua és un conjunt lineal (ja que els seus components ocupen llocs successius dins el conjunt) d'elements en el qual aquests s'introdueixen per un extrem i s'eliminen per l'altre. En conseqüència, els elements d'una cua seran eliminats en el mateix ordre en què es van inserir.

A causa del funcionament d'inserció i eliminació, les cues es coneixen com a estructures FIFO (first in - first out), ja que el primer element que va ser inserit (first in) és el primer en ser eliminat (first out).

La noció de cua ens és familiar: cua en una taquilla d'un cinema, en un taulell d'una botiga... Per a obtenir una entrada ens col·loquem al final de la cua i esperem a arribar a l'inici. Quan ens donen l'entrada, abandonem la cua. En aquest punt cal deixar clar que entenem que hi pugui haver persones que pensin que l'inici de la cua és l'extrem on elles se situen quan s'incorporen a la cua i que el final sigui l'extrem pel qual abandonen la cua. No cal discutir. En tot aquest apartat, considerarem que l'extrem final de la cua és aquell pel qual un element s'incorpora a la cua i l'extrem inici és aquell pel qual un element abandona la cua. D'acord?

Amb les cues solament es poden efectuar dues operacions: `encuar` i `desencuar`. (!!)

Hi ha autors que permeten dues operacions més: `inici` pensada per a consultar l'element que és a punt d'abandonar la cua (l'element que es troba a l'inici de la cua) i `final`, pensada per a consultar el darrer element que ha arribat a la cua (l'element que es troba al final de la cua). Nosaltres permetrem la utilització d'aquestes dues operacions.

Tornem-ho a repetir. Tot i que les implementacions de cua ho permetin, el concepte de tipus de dada cua porta implícit que no s'hi pugui fer altra cosa que les dues (o quatre) operacions esmentades, és a dir, no es pot realitzar un recorregut per una cua, no es pot efectuar una recerca per una cua, no es pot treure l'element del final... (!!)

3.2.1. Implementació del tipus de dada cua

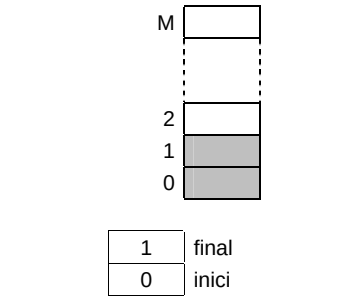
Tradicionalment hi ha dues implementacions del tipus de dada cua:

a) Mitjançant taules

En aquest cas, en la creació de la cua cal definir la grandària màxima de la cua i tenir dues variables enteres per a apuntar l'inici i el final de la cua.

La figura 11 ens mostra la implementació d'una cua mitjançant una taula de grandària M+1 que ja conté dos elements. Observem que la variable `inici` apunta a la casella que conté el primer element incorporat a la cua i la variable `final` apunta a la casella que conté el darrer element incorporat a la fila.

Figura 11. Implementació d'una cua mitjançant una taula



Aquesta implementació té, de manera semblant a les piles, un problema: l'espai reservat de memòria és fix i no pot augmentar ni disminuir.

A més, en les cues, com que interessa un ús eficient de la memòria, cal tractar la taula amb la qual implementem la cua com una estructura circular i fer que les eliminacions deixin espai per a futures insercions. És a dir, suposant que implementem la cua amb una taula de `M+1` posicions, tenim que:

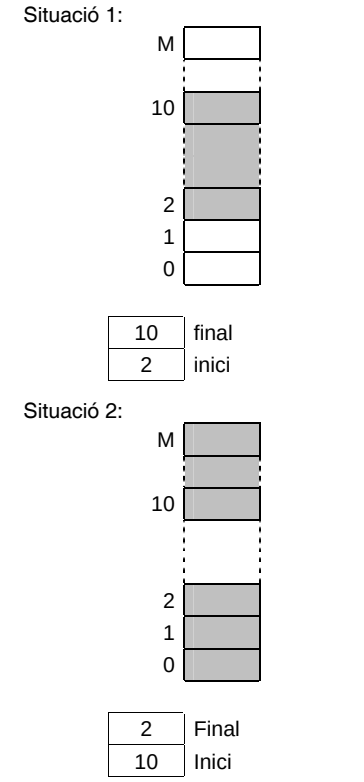
- Si `inici==0`, llavors obligatòriament `final>=inici`, i resulta que la cua està ocupant el tros de taula que va des de la posició 0 fins a la posició `final`. Notem-ho `[0,...,final]`. És el cas que s'observa a la figura 11.

- Si `inici>1`, llavors tenim dues possibilitats respecte al valor de `final`:

– Si `final>=inici`, llavors la cua està ocupant el tros de taula que va des de la posició `inici` fins a la posició `final`. És a dir `[inici,...,final]`. És el cas exemplificat a la situació 1 de la figura 12.

– Si `final<inici`, llavors la cua està ocupant dos trossos de taula: el que va des de la posició `inici` fins a la posició `M` i el que va des de la posició `0` fins a la posició `final`. O sia, `[inici,...,M] U [0,...,final]`. És el cas exemplificat a la situació 2 de la figura 12.

Figura 12. Estat d'una cua, implementada mitjançant una taula, al llarg del temps



b) Mitjançant llistes simplement encadenades (!!)

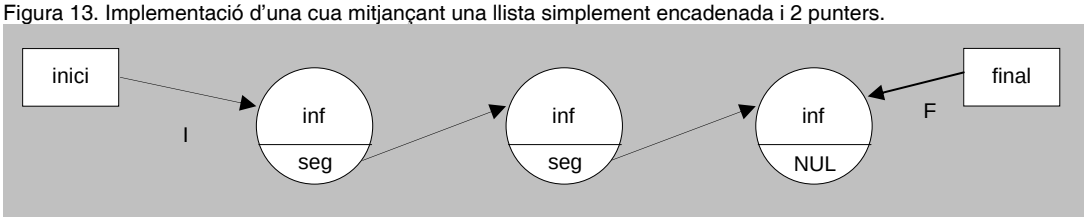
En aquest cas, es tracta de crear una llista simplement encadenada, però amb un segon punter que apunti el darrer element de la llista. La figura 13 ens ho exemplifica. Les dues operacions sobre cues consisteixen en dues de les operacions sobre llistes simplement encadenades:

- Encuar un element és equivalent a afegir un element pel final de la llista (zona F de la figura 13). Aquesta operació podrà ser codificada de manera molt més eficient que en les llistes simplement encadenades amb un únic punter a l'inici (abans calia fer el recorregut de tota la llista per a arribar al final) gràcies a l'aparició del punter al node final.

- Desencuar un element és equivalent a esborrar un element per l'inici de la llista (per la zona I de la figura 13) i recollint l'element eliminat de la cua via paràmetre per referència.

A més, és interessant tenir una operació per a eliminar la cua, la qual és equivalent a l'operació corresponent per a eliminar una llista.

Figura 13. Implementació d'una cua mitjançant una llista simplement encadenada i 2 punters.



En aquesta implementació l'única limitació es troba en la quantitat de memòria disponible. La cua pot créixer i disminuir segons les necessitats. Per tant, és la millor opció.

Com que les dues operacions no seran exactament iguals que en el tractament de llistes simplement encadenades a causa de l'aparició del segon punter, en presentarem el disseny.

```
funció encuar (var inici:^node, var final:^node, element:T) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la cua /* per on es desencua */
      final: Punter que apunta el final de la cua /* per on s'encua */
      element: Dada a afegir a la cua
   Retorna:
      0 : Operació correcta
      1 : Error per manca de memòria
*/
   var aux:^node; fivar
   assignar_memòria (aux);
   si aux == NUL llavors retorna 1; fisi
   aux^.inf = element; aux^.seg = NUL;
   si final == NUL
      llavors inici = aux; final = aux; /* la cua era buida */
      sinó final^.seg = aux; final = aux; /* la cua no era buida */
   fisi
   retorna 0;
fifunció
```

Observeu que, tot i que l'encuament es produeix pel final de la cua, cal passar tots dos punters, `inici` i `final`, per referència. El punter `final` sempre queda modificat després d'una execució de la funció `encuar` excepte si es produeix un error per manca de memòria. En canvi, el punter `inici` quedarà modificat després d'una execució de la funció `encuar` en cas que la cua sigui buida.

```
funció desencuar (var inici:^node, var final:^node, var element:T) retorna natural és
/* Arguments:
      inici: Punter que apunta l'inici de la cua /* per on es desencua */
      final: Punter que apunta el final de la cua /* per on s'encua */
      element: Variable on recollir la dada desencuada
   Retorna:
      0 : Operació correcta
      1 : Error perquè la cua és buida
*/
   var aux:^node; fivar
   si inici == NUL llavors retorna 1; fisi
   aux = inici; element = inici^.inf; inici = inici^.seg;
   si inici == NUL llavors final = NUL; fisi
   alliberar_memòria (aux); retorna 0;
fifunció
```

Observeu que, tot i que el desencuament es produeix per l'inici de la cua, cal passar tots dos punters, `inici` i `final`, per referència. El punter `inici` sempre queda modificat després d'una execució de la funció `desencuar` excepte si la cua era buida. En canvi, el punter `final` quedarà modificat després d'una execució de la funció `desencuar` en cas que la cua només tingui un element. Llavors es convertirà en una cua buida.

Podríem considerar el tipus cua que contingués els dos punters, de la forma següent:

```
tipus cua: tupla
              inici, final: ^node;
           fitupla
```

En aquest cas, les dues funcions per a gestió de cues tindrien les declaracions següents:

```
funció encuar (var c: cua, element:T) retorna natural és
funció desencuar (var c: cua, var element:T) retorna natural és
```

Evidentment cal adequar la definició de les dues funcions als nous arguments. Aquesta manera de fer té un avantatge: l'argument `c` porta incorporats els dos punters a l'inici i al final de la cua. D'aquesta manera, si en un programa es treballa amb diverses cues, no hi ha possibilitat d'equivocar-se en el pas dels paràmetres ni es poden estar barrejant punters a diferents cues.

3.2.2. Aplicacions de les cues

Les cues s'utilitzen molt en informàtica. S'acostumen a aplicar en els processos que necessiten una gestió d'ordre FIFO, com per exemple:

a) Aplicacions informàtiques específiques que gestionen reserves (hotels, avions, trens, preinscripció universitària...).

b) Processos dels mateixos sistemes operatius per a resoldre les demandes d'assignació de recursos en l'explotació diària dels sistemes (sistemes d'impressió, treballs a realitzar, assignació de memòria als processos...).

Tot i que aquest segon camp d'aplicació pugui donar la sensació d'existir només en grans sistemes informàtics, cal tenir en compte que en qualsevol xarxa d'àrea local, per petita que sigui i per infrautilitzada que estigui, en la qual hi hagi alguna impressora compartida, l'ordinador que gestiona la impressora ha de tractar adequadament les peticions d'impressió dels diferents ordinadors de la xarxa. Fins i tot en un sistema monousuari com el DOS hi havia un sistema d'impressió que gestiona les cues d'impressió de les diferents impressores connectades a l'ordinador.