Tipus de Dades, Estructures i Llistes en **C**
Stacks i Cues

Lluís Alsedà

Departament de Matemàtiques
Universitat Autònoma de Barcelona
http://www.mat.uab.cat/~alseda

Versió 1.0 (13 d'abril 2021)

Subjecte a una llicència *Creative Commons Internacional de Reconeixement-NoComercial-CompartirIgual 4.0* (http://creativecommons.org/licenses/by-nc-sa/4.0/)

Continguts

Prerequisits necessaris de **C** 1
Creació de nous tipus de dades i àlies 2
Estructures: creació de nous tipus de dades compostes i complexes .. 6
Organitzacions senzilles d'estructures 20
Cerques eficients per vectors i vectors d'apuntadors 36
Cues 47
Stacks 57

Prerequisits necessaris de C

El material contingut a aquesta presentació usa lliurement continguts de:

Programació en C, http://mat.uab.cat/~alseda/MatDoc/C.pdf

En concret:

- Apuntadors en **C** (pàgines 45 – 70)
- Apuntadors Aplicats (pàgines 71 – 81)
- Cadenes de caràcters (pàgines 82 – 92)
- Ordenació en **C**: la funció `qsort` aprofitant que sabem apuntadors i `strcmp` (pàgines 93 – 101)
- Assignació dinàmica de memòria (pàgines 102 – 108)

Creació de nous tipus de dades i àlies

Índex

1. La declaració `typedef`
2. Exemples

Creació de nous tipus de dades — typedef

El llenguatge **C** disposa d'una declaració anomenada `typedef` que permet la creació de nous tipus de dades.

Declaració

```
typedef tipus_a_definir sinonim;
typedef tipus_a_definir sinonim [dim];
typedef tipus_a_definir sinonim [dim1][dim2];
```


Exemples

```
typedef unsigned int enter;
typedef float Vector [100];
typedef char * string;

enter a, b = 3;
Vector v1, vv[73];
string s;
string *t;
```

`enter` és un sinònim d'`unsigned int`. N'hem creat dues instàncies: `a` (sense inicialitzar) i `b` (inicialitzat a `3`).

`Vector` Els seus objectes són vectors `float` de dimensió 100. Així `v1` és un vector `float` de dimensió 100 sense inicialitzar i `vv` és un vector de mida 73 de vectors float de mida 100 (és a dir una matriu 73×100). `v1` i `vv` s'utilitzen de forma estàndard.
Exemple: `i = v1[19]; v1[20-j] = a;`
`vv[19][97] = b * v1[14];`

`string` Els seus objectes són apuntadors a `char`. `t` és un apuntador a `string` i, per tant, és un apuntador a apuntadors *char*.

Exemple d'ús

```
#include <stdio.h>

typedef unsigned char loopindex; // per bucles de 0 a 255
#define VectorDIM 15
typedef float Vector [VectorDIM];

void main() {
   register loopindex i;
   Vector A;

   for(i=0; i<VectorDIM; i++) A[i] = 2*i;
   for(i=0; i<VectorDIM; i++) printf("%d %f\n", i, A[i]);
}
```


Estructures: creació de nous tipus de dades compostes i complexes

Índex

1. Introducció a les estructures
2. Declaració d'estructures
3. Operacions d'estructures
4. Un exemple bàsic complet
5. Herència: Una estructura pot contenir altres estructures
6. Pas d'estructures a funcions
   - Pas per valor
   - Pas per referència amb apuntadors

Introducció a les estructures

Una estructura és un conjunt d'elements heterogenis (anomenats camps) unificats sota un mateix nom, tipus i espai de memòria.

Les estructures permeten agrupar elements heterogenis en un tipus de dada que els englobi i relacioni.

Declaració d'estructures

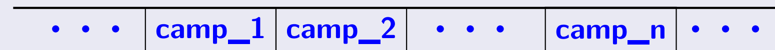
Declaració estàndard

```
struct nom_estructura {
  tipus_1 camp_1;
  tipus_2 camp_2;
         ...
  tipus_n camp_n;
} ;
```

Millor creant un nou tipus de dades que correspongui a aquesta estructura

```
typedef struct [nom_opcional] {
  tipus_1 camp_1;
  tipus_2 camp_2;
         ...
  tipus_n camp_n;
} nom_nou_tipus_de_dada ;
```

Cada instància d'una estructura conté tots els camps de l'estructura



una instància de `struct nom_estructura`
o de `nom_nou_tipus_de_dada`

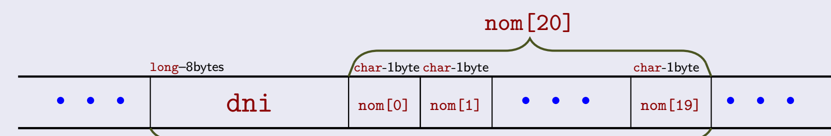
`sizeof(nom_estructura)` =
`sizeof(nom_nou_tipus_de_dada)` =
`sizeof(camp_1)`+`sizeof(camp_2)`+···+`sizeof(camp_n)`

Exemple

```
struct fitxa_alumne {
   long dni;
   char nom [20];
} alumne1;

struct fitxa_alumne fitxes[33];
```

```
typedef struct [nom_opcional] {
   long dni;
   char nom [20];
} fitxa_alumne ;

fitxa_alumne alumne1, fitxes[33];
```

L'estructura de memòria d'`alumne1` i totes les `fitxes[i]`



`sizeof(fitxa_alumne)` =
`sizeof(dni)` + `sizeof(nom[20])` =
`sizeof(long)` + 20 * `sizeof(char)` = 28

Operacions d'estructures

Assignació

Les estructures es poden assignar entre elles, a l'igual que passa amb les variables normals (però no els vectors, ni les cadenes!).

Exemple: `fitxes[10]=fitxes[1];` copia la fitxa de l'alumne 1 al 10 i `fitxes[0]=alumne1;` copia la fitxa de `alumne1` al 0.

Inicialització

Es pot fer dintre el programa on es declara la variable estructura, posant tots els camps entre dues claus i separant-els per una coma.

Exemple:
`fitxa_alumne alumne1 = {35353535, "Pere Primer"};`

Operacions d'estructures (II)

Accés a les dades internes d'una estructura

Per a accedir a una estructura de dades es pot emprar l'operador `.` ("punt").
La sintaxi és molt simple: `nom_estructura.nom_camp`

Exemple: `alumne1.dni = 35353535;`

Exemple: `fitxes[22].dni = 35353535;`

Mida d'una estructura

La mida de l'estructura s'obté emprant l'operador-funció ja conegut: `sizeof()`

Exemple: `sizeof(fitxa_alumne)` o també

Exemple: `int mida = sizeof(alumne1);`

CC BY NC SA

Operacions d'estructures (III)

Apuntadors i aritmètica d'apuntadors

Funcionen exactament igual que per els altres tipus de dades.

Exemple: `fitxa_alumne *f;` declara un apuntador `f` a dades (estructures) del tipus `fitxa_alumne`

Exemple: `f = fitxes + 22;` apunta a `fitxes[22]`. Així:

- `*f` és `fitxes[22]`.
- `f` apunta a la posició de memòria `fitxes + 22*sizeof(fitxa_alumne)`.

CC BY NC SA

Operacions d'estructures (IV)

Accés a les dades internes d'un apuntador a una estructura

Per a accedir als camps d'una estructura des d'un apuntador es poden emprar els operadors `*` i `.` ja coneguts.

Exemple: `fitxes[22].dni = 35353535;`
és totalment equivalent a
    `(*f).dni = 35353535;` i a
    `(*(fitxes+22)).dni = 35353535;`

Hi ha una drecera per la construcció "`* .`": l'operador `->`
La sintaxi és: `apuntador_estructura->nom_camp`
Així els dos exemples anteriors queden, més simples:

Exemple: `f->dni = 35353535;` i
    `(fitxes+22)->dni = 35353535;`

CC BY NC SA

Un exemple bàsic complet

```
/* Trobar la distància entre dos punts */

/* declarem l'estructura punt del pla */
typedef struct { float x, y; } puntpla ;

float dist( puntpla a, puntpla b ){
   float x = a.x-b.x, y = a.y-b.y;
   return sqrt(x*x + y*y);
}

void main() {
  puntpla a, b; //declarem variables de l'estructura creada

  printf("\tOPERANT AMB COORDENADES\n\n");

  printf("\tEntra les coordenades x, y del punt A: ");
  scanf("%f %f", &(a.x), &(a.y)); //definim les coords del punt a

  printf("\tEntra les coordenades x, y del punt B: ");
  scanf("%f %f",&b.x, &b.y); //definim la variable b

  printf("\n\tLa distancia entre els dos punts es %3.2f\n\n", dist(a,b));
}
```

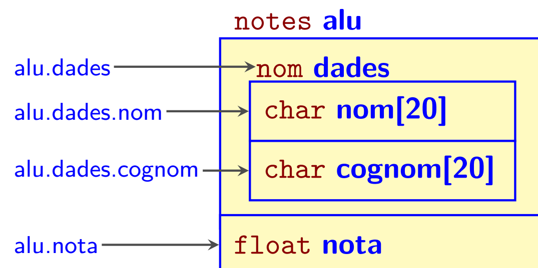

CC BY NC SA

Herència d'estructures: Una estructura pot contenir altres estructures

Les estructures es niuen declarant-les i incloent-les a dins d'una estructura com una dada més. Com amb els altres tipus de dades s'hi accedeix amb l'operador punt.

Declaracions

```
typedef struct {
  char nom[20], cognom[20];
} nom;

typedef struct {
  nom dades;
  float nota;
} notes;

notes alu;
```



Exemple d'herència d'estructures

```
#include <stdio.h>
#define NALUM 20

typedef struct { char nom[20], cognom[20]; } nom;
typedef struct { nom dades; float nota; } notes;

void main() { notes als[NALUM];
  printf("\tNOTES ALUMNES\n");

  for (int i=0; i < NALUM; i++) {
     printf("\tEntra nom, cognom i nota alumne %i: ", (i+1));
     fflush(stdin);
     scanf("%s%s%f", als[i].dades.nom,
                     als[i].dades.cognom,
                     &(als[i].nota)       );
   }

   printf("\n\tEL RESULTAT HA ESTAT:\n");
   for (i=0;i<NALUM;i++) {
       printf("\tAlumne %i:\n", (i+1));
       printf("\t\tNom: %s%s\n",als[i].dades.nom, als[i].dades.cognom);
       printf("\t\tNota: %3.1f\n", als[i].nota);
   }
}
```


Pas d'estructures a funcions

Com amb els tipus de dades bàsics el pas a funcions es pot fer per *valor* o per *referencia*.

En estructures grans és millor fer el pas per referència, ja que passar per valor podria omplir la memòria. Es veurà com es treballa amb dos exemples, que fan servir l'estructura `data` següent:

```
typedef struct {
   int dia, mes, any;
} data ;
```

Per a il · lustrar les dues maneres de passar estructures es definiran dues funcions:

- Calcular una nova data
- Imprimir data

És important entendre com es passen els paràmetres i quins són els resultats en cada cas.

Pas d'estructures a funcions: Pas per valor

No es modifica el contingut de les variables en les funcions

El pas per valor

```
void main() {
   data d2, d1 = {21, 2, 2001};

   imprimir(d1, "d1");
   d2=calcular(d1);
   imprimir(d1, "d1");
   imprimir(d2, "d2");
}
```

Els resultats

```
d1 = 21/2/2001
d1 = 21/2/2001 // No s'ha modificat
d2 = 22/3/2002
```

El codi de les funcions

```
data calcular(data d) { d.dia ++; d.mes ++; d.any ++; return d; }

void imprimir (data d, char v[] ) {
     printf("%s = %i/%i/%i\n", v, d.dia, d.mes, d.any);
}
```

Pas d'estructures a funcions: Pas per referència amb apuntadors

Es modifica el contingut de les variables en les funcions

El pas per referència amb apuntadors (es passa l'adreça de d1)

```
void main() {
   data d2, d1 = {21, 2, 2001};

   imprimir(d1, "d1");
   d2=calcular(&d1);
   imprimir(d1, "d1");
   imprimir(d2, "d2");
}
```

Els resultats

```
d1 = 21/2/2001
d1 = 22/3/2002 // Ara s'ha modificat
d2 = 22/3/2002
```

El codi de la funció `calcular`

```
data calcular(data * d) { d->dia ++; d->mes ++; d->any ++; return *d; }
```


Organitzacions senzilles d'estructures

Índex

1. Vectors d'estructures
2. Vectors d'apuntadors a estructures
3. Introducció a les llistes enllaçades

Metodologia

Tractarem les organitzacions d'estructures mitjançant un exemple: la gestió d'una llista (d'alumnes) amb la que farem les següents operacions base:

- Creació del conjunt adequat d'estructures
- Càrrega de les estructures amb dades d'un fitxer
- Recorregut del conjunt d'estructures: Impressió seqüencial de les dades (per ordre de lectura) realitzant càlculs amb les estructures (per exemple la mitjana de les notes)
- Cerca d'estructures per cadenes de caràcters
- Cerca d'estructures amb criteris numèrics
- Ordenació de la llista usant diversos criteris (`qsort`)
- Eliminació d'un element

Per a fixar idees usarem com a base de dades el fitxer:

`dadesalumnes.dat`

```
Pere;Barniol Serra;3.5;6.6
Joan;Lopez Garcia;8.4;5.5
Ramon;Marti Perez;6.3;7.4
Maria;Barniol Roca;9.5;7.4
```


Vectors d'estructures

Avantatges

- Simplicitat

Inconvenients

- Cada operació es realitza sobre tota l'estructura
- Alenteix el procés si les estructures són molt grans (`sizeof`)

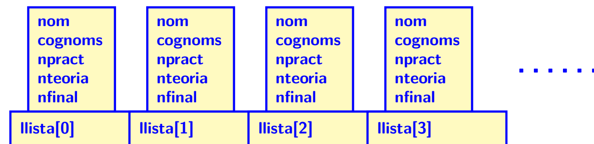
Vectors d'estructures: Un exemple

Declaració de l'estructura

```
typedef struct {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
} alumne;
```

El vector d'estructures

```
alumne *llista;

if ((llista = (alumne *) malloc (nalu*sizeof (alumne))) == NULL){
    fprintf (stderr,
        "\nERROR: No es possible crear la llista d'alumnes ...\n\n"
    );
    return 1;
}
```



Vectors d'estructures: Un exemple (II)

Funcions auxiliars; Declaracions i obertura del fitxer

```
#define NotaFinal(t,p) (0.6 * t + 0.4 * p)
void LlistaAlumne(alumne curr){
   printf("%s, %s. Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",
        curr.cognoms, curr.nom, curr.nteoria, curr.npract, curr.nfinal );
}

int ComparaNota(const void * a, const void * b) {
    return (((alumne*)b)->nfinal - ((alumne*)a)->nfinal);
}

int main () {
  alumne *llista;
  FILE *fin;
  unsigned int i, nalu=0;
  char fitxer[]="dadesalumnes.dat";

// Obrim el fitxer
  if ((fin = fopen (fitxer, "r")) == NULL){ fprintf (stderr,
              "\nERROR: El fitxer '%s' no existeix o no es pot obrir...\n\n",
      fitxer);
      return 1;
  }
```

D'entrada, ambdós paràmetres són apuntadors a `void` i llavors `*a` i `*b` no són cap estructura (en particular no són cap estructura del tipus `alumne`).
En conseqüència, `a->nfinal` i `b->nfinal` no tenen sentit.
Això s'arregla forçant el tipus d'`a` i `b` a `(alumne*)a` i `(alumne*)b`.

Vectors d'estructures: Un exemple (III)

```
// Contem el nombre d'alumnes. Caldria controlar que no és zero
   while (!feof (fin)) if(fgetc(fin) == '\n') nalu++;

// Creem llista d'estructures
   if ((llista = (alumne *) malloc (nalu*sizeof (alumne))) == NULL){
      fprintf (stderr,
         "\nERROR: No es possible crear la llista d'alumnes ...\n\n");
      return 1;
   }
   rewind(fin);
   for(i=0 ; i < nalu ; i++) {
// Carreguem estructura amb dades de fitxer
     fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", llista[i].nom,
             llista[i].cognoms, &(llista[i].nteoria), &(llista[i].npract) );
     llista[i].nfinal = NotaFinal( llista[i].nteoria, llista[i].npract );
   } ; fclose (fin);

// Imprimim dades seqüencials i calculem les mitjanes
{ float m[3] = {0.0, 0.0, 0.0 };
  printf("\n***** Llistat d'alumnes segons l'ordre de lectura:\n");
  for(i=0 ; i < nalu ; i++) {
    printf("Alumne %u: ", i+1); LlistaAlumne(llista[i]);
    m[0] += llista[i].nteoria; m[1] += llista[i].npract; m[2] += llista[i].nfinal;}
  printf("Hi ha %u alumnes\n", nalu);
  printf("Mitjanes: Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",
        m[0]/nalu,m[1]/nalu,m[2]/nalu);
}
```


Vectors d'estructures: Un exemple (IV)

```
/* Cerca d'estructures per cognom */
{  char cognom[]="Barniol";
   printf("\n* Llistat d'alumnes amb cognom %s:\n", cognom);
   for(i=0 ; i < nalu ; i++)
       if(! strncmp(llista[i].cognoms, cognom, strlen(cognom)))
            LlistaAlumne(llista[i]); }

/* Cerca d'estructures per nota */
{  float notamin=7.0;
   printf("\n* Llistat d'alumnes amb nota mínima %f:\n", notamin);
   for(i=0 ; i < nalu ; i++)
       if(llista[i].nfinal >= notamin) LlistaAlumne(llista[i]); }

// Ordenació. qsort. No recomanat: estem movent estructures.
  printf("\n* Llistat d'alumnes segons l'ordre de nota final:\n");
  qsort(llista, nalu, sizeof(alumne), ComparaNota);
  for(i=0 ; i < nalu ; i++) LlistaAlumne(llista[i]);

/* Borrem l'alumne 2 per ordre de nota */
{  int borralu=1;
   nalu--;
   for(i=borralu ; i < nalu ; i++) llista[i]=llista[i+1];
   printf("\n* Alumne 2 eliminat. ");
   printf("Llistat dels alumnes que queden:\n");
   for(i=0 ; i < nalu ; i++) LlistaAlumne(llista[i]); }
```

La instrucció `nalu--` fixa el rang actual del vector `llista` de `0` a `nalu-1` (sense alliberar memòria; és a dir, amb el valor nou de `nalu`, `llista[nalu]` encara existeix però ocupant memòria).
El bucle copia les estructures amb índex més gran o igual que `borralu+1` a la posició anterior; esborrant així la informació de `llista[borralu]` (i deixant `llista[nalu-1] = llista[nalu]`).
És crucial que `nalu` s'hagi disminuït abans del bucle per a evitar accedir al vector `llista` fora de rang.

Vectors d'estructures: Resultats de l'exemple

```
***** Llistat d'alumnes segons l'ordre de lectura:
Alumne 1: Barniol Serra, Pere. Teor = 3.50; Pract = 6.60; Final = 4.74
Alumne 2: Lopez Garcia, Joan. Teor = 8.40; Pract = 5.50; Final = 7.24
Alumne 3: Marti Perez, Ramon. Teor = 6.30; Pract = 7.40; Final = 6.74
Alumne 4: Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66
Hi ha 4 alumnes
Mitjanes: Teor = 6.93; Pract = 6.72; Final = 6.84

***** Llistat d'alumnes amb cognom Barniol:
Barniol Serra, Pere. Teor = 3.50; Pract = 6.60; Final = 4.74
Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66

***** Llistat d'alumnes amb nota mínima 7.000000:
Lopez Garcia, Joan. Teor = 8.40; Pract = 5.50; Final = 7.24
Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66

***** Llistat d'alumnes segons l'ordre de nota final:
Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66
Lopez Garcia, Joan. Teor = 8.40; Pract = 5.50; Final = 7.24
Marti Perez, Ramon. Teor = 6.30; Pract = 7.40; Final = 6.74
Barniol Serra, Pere. Teor = 3.50; Pract = 6.60; Final = 4.74

***** Alumne 2 eliminat. Llistat sequencial dels alumnes que queden:
Barniol Roca, Maria. Teor = 9.50; Pract = 7.40; Final = 8.66
Marti Perez, Ramon. Teor = 6.30; Pract = 7.40; Final = 6.74
Barniol Serra, Pere. Teor = 3.50; Pract = 6.60; Final = 4.74
```


Vectors d'apuntadors a estructures

Avantatges

- Per realitzar operacions no cal moure tota l'estructura. Això és especialment útil a les ordenacions, en les que hi ha moviment massiu de dades
- Costa menys afegir (`realloc`) i esborrar elements
- Es pot usar per crear índexs addicionals de vectors d'estructures

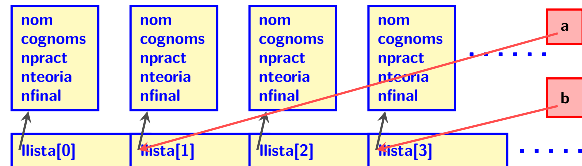
Inconvenients

- Una mica més complicat de programació
- Fa servir una mica més de memòria

Vectors d'apuntadors a estructures: Un exemple

Estructura (com abans)

```
typedef struct {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
} alumne;
```



Funcions auxiliars i declaracions (ometem l'obertura del fitxer)

```
#define NotaFinal(t,p) (0.6 * t + 0.4 * p)

void LlistaAlumne(alumne *curr){
   printf("%s, %s. Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",
     curr->cognoms, curr->nom, curr->nteoria, curr->npract, curr->nfinal);
}

int ComparaCognoms(const void * a, const void * b) {
  return strcmp( (*((alumne**)a))->cognoms, (*((alumne**)b))->cognoms);
}

int main () {
  alumne **llista;
  FILE *fin;
  unsigned int i, nalu=0;
```

Ara usem l'operador `->` perquè `curr` és un apuntador.

`strcmp` compara dos "strings" i el seu valor de retorn coincideix amb el demanats per `qsort`: Negatiu (respectivament, zero o positiu) segons que el primer sigui més petit (respectivament, igual o més gran) que el segon.

Aquí `a` i `b` es declaren com apuntadors de tipus *void* però, en realitat, són apuntadors a algun element del vector `llista` (posem que `a` és un apuntador `void` a `llista[1]`).
Per accedir als cognoms de l'alumne apuntat per `llista[1]`, en primer lloc cal forçar el tipus d'`a` a un apuntador del mateix tipus que `llista` (que ha estat declarat com `alumne **llista`). Aquest apuntador és `(alumne**)a` (que apunta a `llista[1]` amb el tipus correcte).
Llavors, `*((alumne**)a) = llista[1]` i, per tant, s'accedeix al camp `cognoms` de l'alumne apuntat per `llista[1]` amb `(*((alumne**)a))->cognoms`.

Vectors d'apuntadors a estructures: Un exemple (II)

```
// Contem el nombre d'alumnes. Caldria controlar que no és zero
   while (!feof (fin)) if(fgetc(fin) == '\n') nalu++;

// Creem llista d'apuntadors a estructures
   if ((llista = (alumne **) malloc (nalu*sizeof (alumne *))) == NULL){
       fprintf (stderr, "\nERROR: No es possible crear la llista d'alumnes ...\n\n");
       return 1;
   }

   rewind(fin);
   for(i=0 ; i < nalu ; i++) {
     if ((llista[i] = (alumne *) malloc(sizeof (alumne))) == NULL){
       fprintf (stderr, "\nERROR: No es possible crear alumne %u ...\n\n", i);
       return 1;
     }

// Carreguem estructura amb dades de fitxer
     fscanf (fin, "%[a-zA-Z'. ];%[a-zA-Z'. ];%f;%f\n", llista[i]->nom,
             llista[i]->cognoms, &(llista[i]->nteoria), &(llista[i]->npract) );
     llista[i]->nfinal = NotaFinal( llista[i]->nteoria, llista[i]->npract );
   }
   fclose (fin);

// Imprimim dades seqüencials i calculem les mitjanes
{ float m[3] = {0.0, 0.0, 0.0 };
  printf("\n***** Llistat d'alumnes segons l'ordre de lectura:\n");
  for(i=0 ; i < nalu ; i++) {
    printf("Alumne %u: ", i+1); LlistaAlumne(llista[i]);
    m[0] += llista[i]->nteoria; m[1] += llista[i]->npract; m[2] += llista[i]->nfinal; }
  printf("Hi ha %u alumnes\n", nalu);
  printf("Mitjanes: Teor = %#.2f; Pract = %#.2f; Final = %#.2f\n",m[0]/nalu,m[1]/nalu,m[2]/nalu);
}
```

Aquí `llista` és un vector d'elements `alumne *` (és a dir, d'apuntadors a `alumne`).
Per això `llista` és un apuntador doble a `alumne`.

L'estructura `llista[i]`, ara, no existeix. L'hem de crear cada vegada.

`llista[i]` és un apuntador a `alumne`.

A partir d'ara usem l'operador `->` perquè `llista[i]` és un apuntador.

Vectors d'apuntadors a estructures: Un exemple (III)

```
/* Cerca d'estructures per cognom */
{   char cognom[]="Barniol";
    printf("\n***** Llistat d'alumnes amb cognom %s:\n", cognom);
    for(i=0 ; i < nalu ; i++)
        if(! strncmp(llista[i]->cognoms, cognom, strlen(cognom))) LlistaAlumne(llista[i]);
}

/* Cerca d'estructures per nota*/
{   float notamin=7.0;
    printf("\n***** Llistat d'alumnes amb nota mínima %f:\n", notamin);
    for(i=0 ; i < nalu ; i++) if(llista[i]->nfinal >= notamin) LlistaAlumne(llista[i]);
}

// Ordenació. qsort. Ara estem movent apuntadors (de mida molt més petita que les estructures)
  printf("\n***** Llistat d'alumnes per cognoms:\n");
  qsort(llista, nalu, sizeof(alumne *), ComparaCognoms);
  for(i=0 ; i < nalu ; i++) LlistaAlumne(llista[i]);

{   int borralu=1;
    free(llista[borralu]);
    nalu--; for(i=borralu ; i < nalu ; i++) llista[i]=llista[i+1];
    llista[nalu]=NULL;
    printf("\n***** Alumne 2 eliminat. Llistat alfabetic dels alumnes que queden:\n");
    for(i=0 ; i < nalu ; i++) LlistaAlumne(llista[i]);
}
```

Veure el diagrama de l'ordenació a la plana següent.

No és imprescindible. Per desapuntar la darrera posició del vector.

El procés d'esborrat és anàleg a l'anterior.
Ara l'estructura `*llista[borralu]` es pot eliminar ja que no s'usa; això millora el procés anterior (allà no podíem alliberar l'estructura que no s'utilitzava).
Per altra banda

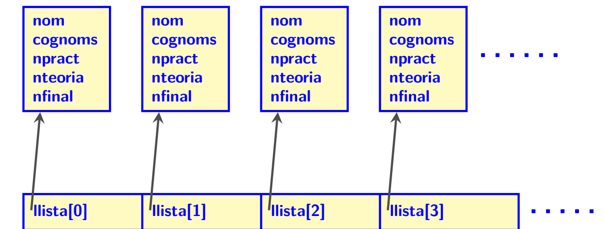
`llista[nalu] = llista[nalu_original-1]`

no s'utilitza però encara ocupa memòria (solament la d'un apuntador).

Vectors d'apuntadors a estructures: Un exemple
Idea de l'ordenació

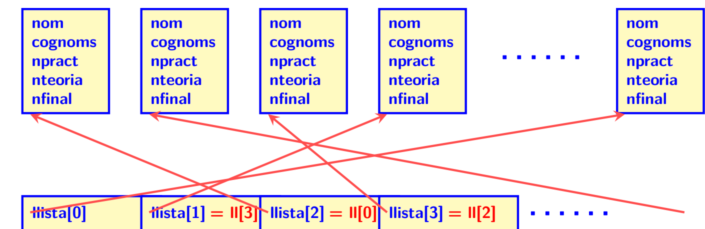
Amb l'ajut de `qsort` passem de:

situació de `llista` abans d'ordenar



a

situació de `llista` després d'ordenar



Una primera aproximació a les llistes enllaçades: estructures que apunten a altres estructures

Avantatges

- Organització de dades estructurada
- La programació és independent del número de dades
- Màxima eficiència en afegir i esborrar elements

Inconvenients

- Usen més memòria que els vectors i tanta memòria com els vectors d'apuntadors
- Programació més complicada
- Cerca eficient molt difícil. És gairebé obligada la cerca seqüencial.
- Ordenació més complicada. Es pot simplificar afegint un "vector d'índexs" auxiliar d'apuntadors a les estructures de la llista.

Que és una llista enllaçada?

Una llista enllaçada és una col · lecció d'elements disposats seqüencialment que permet la inserció i eliminació d'elements en qualsevol lloc de la seqüència.

En una llista enllaçada podem inserir i eliminar nodes sense haver de conèixer la mida de la llista i de les dades. L'eina que permet el funcionament d'aquest mecanisme és la de reservar i alliberar els espais de memòria dels nodes mitjançant l'assignació dinàmica.

En l'ús de les llistes dinàmiques cal sempre tenir present fer la reserva de memòria com a pas previ a la creació d'un node i d'alliberar-la en el moment en què un node desapareix.

Les llistes enllaçades poden ser simples i dobles. En una llista simple, de cada node solament podem saltar al següent. Per tant, solament podem recórrer la llista endavant. En una llista doblement enllaçada, de cada node podem saltar al posterior i a l'anterior. Per tant, podem recórrer la llista tan endavant com endarrere.

En el que segueix, per simplicitat, solament considerarem les llistes enllaçades simples.

Introducció a les llistes enllaçades

Considerem la següent millora de l'exemple `alumne` anterior. Volem crear una llista ordenada del 4 alumnes `B`, `D`, `C`, `A`. Suposem que l'ordre és `B -> D -> C -> A` i que les estructures ja tenen les dades (llegides d'un fitxer o de teclat).

Declaració de l'estructura

```
typedef struct ALUDATA {
  char nom[20], cognoms[40];
  float npract, nteoria, nfinal;
  struct ALUDATA *seg;
} alumne;
```

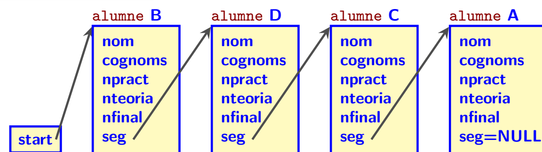
El codi

```
alumne A, B, C, D, *start=NULL;
start = &B;
B.seg = &D;
D.seg = &C;
C.seg = &A;
A.seg = NULL;
```

Nota

Si en una estructura volem incloure un apuntador a estructures del mateix tipus no es pot usar el tipus creat per `typedef`. Cal usar la declaració `struct 'namespace'` com hem fet a la declaració anterior.

L'apuntador `start` és fonamental. Necessitem l'adreça del primer element (un punt d'entrada).



Cerques eficients per vectors i vectors d'apuntadors

Índex

1. Cerca seqüencial versus cerca binària
2. Cerca binària
3. Exemple d'implementació per vectors
4. Exemple d'implementació per vectors d'apuntadors

Cerca seqüencial versus cerca binària

Avantatges

- La cerca seqüencial és *terriblement* ineficient amb grans quantitats de dades ($\mathcal{O}(n)$).
- Les cerques binàries incrementen espectacularment l'eficiència ($\mathcal{O}(\log_2(n))$).

Inconvenients

- Programació més complicada
- Solament funciona en dades ordenades amb el mateix criteri de busca

Introducció a la cerca binària

En informàtica, una *cerca binària* o *algorisme de busca de bisecció* troba la posició d'un valor especificat d'entrada (la *clau* de cerca) dins d'un *vector ordenat en ordre creixent respecte dels valors de la clau*.

És un *algorisme dicotòmic del tipus divideix i venceràs*.

Consisteix a aplicar l'algorisme de bisecció a la cerca de valors. Cal tenir les dades ordenades i indexades.

La cerca binària redueix a la meitat el nombre d'elements a comprovar a cada iteració, de manera que la localització d'una clau (o la determinació de la seva absència) es fa en temps logarítmic ($\log_2(n)$).

Introducció a la cerca binària: Idea de l'algorisme

A cada pas, l'algoritme compara el valor de la clau de cerca amb el valor de de l'element mitjà del vector.

Si les claus coincideixen, llavors s'ha trobat un dels elements buscat i es retorna el seu índex o posició dins del vector.

Altrament, si la clau de cerca és menor que la clau de l'element mitjà, llavors l'algorisme repeteix la seva acció sobre el sub-vector a l'esquerra de l'element mitjà o, si la clau de cerca és més gran, en el sub-vector a la dreta de la clau de cerca.

Si després d'algunes iteracions l'interval de vector que queda per buscar és buit, la clau no pertany al vector i es retorna una indicació especial de `no_trobat`.

Introducció a la cerca binària

Exemple

- La llista on volem cercar és `L = {1 3 4 6 8 9 11}`
- El valor que volem trobar és: `clau = 4`
- Iteracions de l'algorisme:
  1. Inicialitzem l'interval de cerca:
     `inici = 1` (`L[1] = 1`); `final = 7` (`L[7] = 11`).
  2. `punt mig = 4` (`L[4] = 6`).
     Comparar `clau` amb `L[4] = 6`: **`clau` és menor**.
     En funció d'això redefinim l'interval de cerca:
     `inici = 1` (`L[1] = 1`); `final = 3` (`L[3] = 4`).
  3. `punt mig = 2` (`L[2] = 3`).
     Comparar `clau` amb `L[2] = 3`: **`clau` és més gran**.
     En funció d'això tornem a redefinir l'interval de cerca:
     `inici = 3`; `final = 3` (`L[3] = 4`).
  4. `punt mig = 3` (`L[3] = 4`).
     Comparar `clau` amb `L[3] = 4`: **Són iguals**.
     Ja hem acabat i retornem `index = 3`.

Cerca binària

Algorisme: de cerca binària amb interval com a rang de cerca

```
procedure BINARY_SEARCH(Vector, Vlen, clau)
    imin ← 0; imax ← Vlen − 1;          ▷ Inicialització de l'interval de cerca (tot el vector)
    while imax ≥ imin do                 ▷ Iteració: continuar buscant mentre [imin, imax] no està buit
        imid ← (imin+imax)/2;            ▷ Punt mig
        if Vector[imid] = clau then
            return imid;                 ▷ imid és l'element buscat. Retornem el seu índex
        else if Vector[imid] < clau then ▷ clau és a la dreta d'imid
            imin ← imid + 1              ▷ El nou interval serà [imid+1,imax]
        else                             ▷ clau és a l'esquerra d'imid
            imax ← imid − 1              ▷ El nou interval serà [imin,imid-1]
        end if
    end while
    return −1;                           ▷ No s'ha trobat la clau. Retornem -1 que no és un índex vàlid.
end procedure
```

Observacions

- *Solament funciona amb dades ordenades amb el mateix criteri que la busca.*
- A cada iteració `imax` (`imin`) és més petit (gran) o igual que a la iteració anterior. Per tant, dins del bucle, $0 \leq$ `imin` $\leq$ `imax` $\leq$ `Vlen` $- 1$ (en particular, ni `imin` ni `imax` poden sortir del rang del vector). Això justifica la sortida d'error fora del bucle.

Cerca binària: exemples d'implementació

La funció de cerca binaria numèrica
per un vector `unsigned long` ordenat de petit a gran

```c
unsigned long BinarySearch(unsigned long key, unsigned long *list, unsigned long lenlist){
  register unsigned long start=0UL, afterend=lenlist, middle, try;

  while(afterend > start){ middle = start + ((afterend-start-1)>>1); try = list[middle];
    if (key == try) return middle;
    else if ( key > try ) start = middle+1;
    else afterend = middle;
  }
  return ULONG_MAX;
}
```

Com es crida la funció

```c
unsigned long index = BinarySearch(key, llista, numElements);
if(index == ULONG_MAX) printf("ERROR: Alumne no trobat\n");
else printf("Key %lu has index %lu\n", key, index);
```

Observacions

- **`afterend` en lloc d'`end`**
  Mantenir la posició següent al final de l'interval en lloc de la posició final té diversos avantatges:
  - **`while(afterend > start)`**: Permet fer la comparació `>` en comptes de `>=` (més complicada).
  - **`afterend=middle`**: Evita fer `end = middle-1`, que pot donar negatiu (desastrós: usem `unsigned`'s!!).
- **`middle = start + ((afterend-start-1)»1)` = `start + ((afterend-start-1)/2)`**
  (**`(afterend-start-1)»1`** és solament un forma ràpida de dividir un enter per 2)
  La fórmula `middle = (start + afterend -1)/2` no es pot usar quan `start`,`afterend`>`ULONG_MAX/2` ja que `start+afterend-1`>`ULONG_MAX` i tenim un overflow.
- **`try = list[middle];`**
  Per evitar accedir a la memòria dues vegades
- **`return ULONG_MAX;`**
  No podem tornar `-1` com a codi d'error ja que retornem `unsigned`'s. En lloc de `-1` triem l'`unsigned long` més gran. Això implica que no podem usar vectors d'aquesta mida (no és gaire preocupant donat que `ULONG_MAX` $= 2^{64} - 1 = 18.446.744.073.709.551.615$).

Cerca binària: exemples d'implementació

La funció de cerca binaria numèrica
per un vector d'estructures ordenat de gran a petit

```
long BuscaNotaCercaBinaria(alumne *llista, unsigned int nalu, const float nota){
  register unsigned int start=0UL, afterend=nalu, middle;
  register float try;

  while(afterend > start){
    middle = start + ((afterend-start-1)>>1);
    try = llista[middle].nfinal;
    if (fabs(try - nota) <= 5.0e-3) return (long) middle;
    else if ( nota < try ) start = middle+1;
    else afterend = middle;
  }
  return -1L;
}
```

Com es crida la funció

```
long index = BuscaNotaCercaBinaria(llista, nalu, 7.24);
if(index == -1) printf("ERROR: Alumne no trobat\n");
else LlistaAlumne(llista[index]);
```

Observacions

- **`fabs(try - nota) <= 5.0e-3`**
  Recordem que en punt flotant no té sentit fer comparacions `try == nota`.
- **`return -1L;`**
  Noteu que ara retornem `long` (amb signe — `long BuscaNotaCercaBinaria(alumne ...)`. Per tant podem tornar valors negatius (fora del rang d'índexs d'un vector). No hi ha cap problema en fer això ja que `UINT_MAX` = 4294967295U i `LONG_MAX` = 9223372036854775807L. Per tant, una variable `long` pot contenir qualsevol `unsigned int`.

Cerca binària: exemples d'implementació

La funció de cerca binaria alfabètica
per un vector d'apuntadors a estructures ordenat de petit a gran

```
long BuscaNomCercaBinaria(alumne **llista, unsigned int nalu, const char *cognom){
    register unsigned int start=0UL, afterend=nalu, middle;
    register unsigned short cognomlen = strlen(cognom);
    register int rescom;

  while(afterend > start){ middle = start + ((afterend-start-1)>>1);
    rescom = strncmp(llista[middle]->cognoms, cognom, cognomlen);
    if (rescom == 0) return middle;
    else if ( rescom < 0 ) start=middle+1;
    else afterend=middle;
  }
  return -1;
}
```

Per estalviar *moltes vegades* aquest càlcul dins del bucle.

Comparació guardada a la variable `rescom` per a evitar accedir a la memòria dues vegades.

La crida a la funció

```
long index = BuscaNomCercaBinaria(llista, nalu, "Lopez");
if(index == -1) printf("ERROR: Alumne no trobat\n");
else LlistaAlumne(llista[index]);
```


El C també té la cerca binària de llibreria: `bsearch`

La funció `bsearch` té capçalera

```
void * bsearch(const void *key, const void *base, size_t nmemb,
               size_t size, int (*compara)(const void *, const void *));
```

La funció `bsearch` que fa?

Busca `key` dins del vector `base` (format per `nmemb` blocs de mida `size` bytes) fent servir l'algorisme de *cerca binària*.

La funció torna un apuntador a l'element buscat (*no el seu índex dins el vector*) o `NULL` si no l'ha trobat.

La funció `bsearch` requereix que el *vector* `base` *estigui ordenat amb el mateix criteri de la busca*.

El funcionament de `bsearch` es basa en la funció `compara` exactament igual que la funció `qsort`, excepte que ara la funció `compara` compara un apuntador `void` a `key` amb un apuntador `void` a un dels *tokens* del vector `base`.

`bsearch` exemple

Ús de la funció `bsearch`
per un vector d'estructures ordenat numèricament de gran a petit

```
float nota = 7.24;
alumne *alu = bsearch(&nota, llista, nalu, sizeof(alumne), ComparaKeyNota);
if(alu == NULL) printf("ERROR: Alumne no trobat\n"); else LlistaAlumne(*alu);
```

La funció de comparació

```
int ComparaKeyNota(const void * key, const void * b) {
  if ( fabs(((alumne*)b)->nfinal - *(float*)key) <= 5.0e-3 ) return 0;
    else if ( *(float*)key <  ((alumne*)b)->nfinal) return 1;
    else return -1;
}
```

Ús de la funció `bsearch`
per un vector d'apuntadors a estructures ordenat alfabèticament de petit a gran

```
char cognom[] = "Lopez";
alumne **alu = bsearch(cognom, llistaAlumnes, nalu, sizeof(alumne*), ComparaKeyCognom);
if(alu == NULL) printf("ERROR: Alumne no trobat\n");
else LlistaAlumne(*alu);
```

La versió de `LlistaAlumne` per vectors d'apuntadors accepta un apuntador a alumne (`alumne *`).

La funció de comparació

```
int ComparaKeyCognom(const void * key, const void * b) {
    return strcmp(key, (*((alumne**)b))->cognoms);
}
```

Cues

Índex

1. Introducció i motivació: Aplicacions de les cues
2. Implementació d'una cua de mida arbitrària amb una llista enllaçada

Introducció a les cues

Una cua és una estructura *First In – First Out (FIFO)* on la primera dada en entrar és la primera en sortir:

- Un element es pot inserir en qualsevol moment a la cua, però només l'element que ha estat més temps a la cua pot ser retirat.
- Els elements s'insereixen a la part posterior (en cua) i es retiren per la part davantera.

Un exemple d'aquesta estructura són les cues de la vida real (al supermercat, . . .).

Aplicacions de les cues

- *Sistemes operatius*: gestió seqüencial de processos i distribució de tasques. Per exemple: cues d'impressió, distribució dels temps de CPU,. . .
- *Scheduling*: programació de tasques (per exemple l'ordre en que un viatjant de comerç fa les visites), distribució de tasques en diferents processadors (màquines – això es fa amb diverses cues; per exemple una per màquina). La distribució de temps de CPU és un cas particular d'això i els horaris d'una escola per aules (cada aula es considera una màquina) també.
- Procés seqüencial de dades en temps real.
- Simulació de processos que involucrin cues a fi de millorar-ne l'eficiència.

Una important aplicació de les cues (que ens serveix com exemple il · lustratiu d'implementació) és la programació no recursiva de moviments en profunditat en grafs.

Grafs: Definicions i Algorismes Bàsics,
`http://mat.uab.cat/~alseda/MatDoc/GrafsDefimovs.pdf`

Funcions fonamentals de les cues

encua Insereix un objecte a la part posterior de la cua

desencua Si la cua no està buida en retira el seu primer element. En cas contrari torna un codi d'error.

Així mateix una cua pot tenir associades funcions auxiliars. Entre elles:

mida retorna el nombre d'objectes a la cua

CuaEsBuida retorna un valor booleà que indica si la cua està buida o no

A més es necessita un ***tipus de dades específic***.

Implementació amb vectors de mida fixada versus llistes enllaçades de mida arbitrària

Avantatges dels vectors

- Simplicitat de programació

Desavantatges dels vectors

- Cal reposicionar constantment els elements de la cua en desencuar elements: molt ineficient
- Si la cua s'omple cal augmentar la mida del vector contenidor i copiar tots els elements de la cua del contenidor vell al nou: ineficient

Avantatges de les llistes

- La mida de la cua està limitada solament per la quantitat de memòria lliure de l'ordinador
- La programació és independent del número d'elements de la cua
- Màxima eficiència en afegir i esborrar elements

Desavantatges de les llistes

- Usen més memòria que els vectors encara que aquesta s'ajusta automàticament a la mida de la cua

Implementació d'una cua amb una llista enllaçada Declaracions i tipus de dades específics

Com sempre cal definir un tipus de dada específic que correspongui als elements individuals de la llista (cua).

Com que en una cua volem accedir-hi pel principi i pel final (amb sengles apuntadors), i també és útil saber el nombre d'elements de la cua, es recomana definir un tipus de dades específic pel *contenidor cua* que ajunti tots aquests elements de la cua.

Declaració dels elements de la cua

```
typedef struct ElementDeCua {
    unsigned int h;
    struct ElementDeCua *seg;
} CuaElem;
```

El contenidor de la cua

```
typedef struct {
   CuaElem *start, *end;
   unsigned nel;
} Cua;
```

Nota: `h` és la variable destinada a contenir la informació identificativa de l'element que es posa a la cua.

Conveni: Es fonamentar fixar i mantenir el conveni que la cua és buida si i només si `start = end = NULL` i `nel = 0U`.

Implementació d'una cua amb una llista enllaçada Funcions base

Funcions senzilles de gestió de la cua

```
void IniCua (Cua *Q) {
     Q->start = Q->end = NULL;
     Q->nel = 0U;
}
int CuaEsBuida( Cua Q ){ return ( Q.start == NULL ); }
int CuaEsBuida( Cua Q ){ return ( Q.nel == 0U ); }
CuaElem * top( Cua Q ) { return Q.start; }
```

Aquesta funció es l'encarregada de facilitar el manteniment del conveni. Notis que la funció ha de modificar el contenidor cua. Per tant, és obligatori passar un apuntador al contenidor cua a la funció. Llavors, els elements del contenidor cua es referencien `Q->`.

Versió alternativa de la funció `CuaEsBuida`

Retorna l'adreça de l'element al principi de la cua (no el seu contingut).

Ús de les funcions de gestó de la cua

Inicialització

```
Cua LaCua;
IniCua(&LaCua);
```

La funció `top` permet recuperar o modificar dades del primer de la cua

```
htotal += top(LaCua)->h;
top(LaCua)->h = 2.0*hnou + 4;
```


Implementació d'una cua amb una llista enllaçada Recorregut seqüencial d'una llista

`AlliberaCua`: recorregut seqüencial per esborrar la llista

```
void AlliberaCua( Cua *Q ){ CuaElem *llistaindexaux = Q->start;
   while(llistaindexaux){ CuaElem *node_actual = llistaindexaux;
     llistaindexaux = llistaindexaux->seg;
     free (node_actual);
   }
   IniCua(Q);
}
```

Per reconstruir l'assignació del conveni ja que hem buidat la cua

`AlliberaCua` en versió més clara i compacta (però menys eficient)

```
void AlliberaCua( Cua *Q ){
   while( !CuaEsBuida(*Q) ) desencua(Q);
   IniCua(Q);
}
```

Exercici (per lliurament suplementari)

Programa una funció `ImprimeixCua` que imprimeixi seqüencialment tots els elements de la cua.

Implementació d'una cua amb una llista enllaçada
la funció desencua
Esborrant el node inicial de la cua

desencua

Per poder modificar el contenidor cua

```
unsigned int desencua( Cua *Q ){
  if( CuaEsBuida(*Q) ) return UINT_MAX;
  CuaElem *node_inicial = Q->start;
  unsigned int h_inicial = node_inicial->h;
  Q->start = Q->start->seg;
  free(node_inicial);
  Q->nel--;
  return h_inicial;
}
```

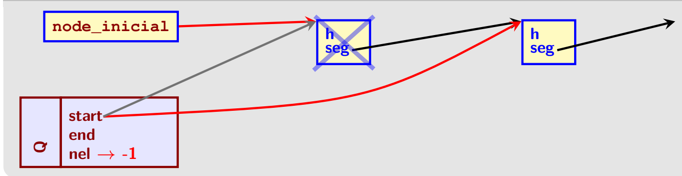
Codi d'error compatible amb `unsigned int`

Quan buidem la cua (en esborrar el darrer i únic element) reconstruïm l'assignació del conveni `Q->start = NULL` i `Q->nel = 0` (en canvi no tenim `Q->end = NULL`).

Exemple d'ús

```
h = desencua(&LaCua);
```

desencua en imatges



Implementació d'una cua amb una llista enllaçada
la funció encua
Afegint un node al final de la cua

encua

Per poder modificar el contenidor cua

Inici:
* Crear l'estructura aux
* Omplir-la amb les dades: `aux->h=h;`
* Inicialitzar apuntador seg: `aux->seg=NULL;`

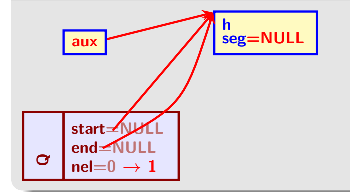
```
int encua( unsigned int h, Cua *Q ){
  CuaElem *aux = (CuaElem *) malloc(sizeof(CuaElem));
  if( aux == NULL) return 0;

  aux->h = h; aux->seg = NULL;

  if( Q->start ) Q->end->seg=aux;  else Q->start=aux;
  Q->end = aux; Q->nel++;
  return 1;
}
```

Part obligatòria de l'addició del node

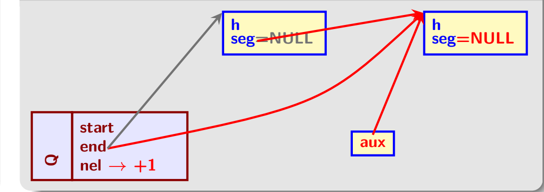
Exemple d'ús

```
if (! encua(10, &LaCua))){
  fprintf (stderr,
    "\nERROR: Cua overfull\n"
  );
  return 11;
}
```

Cas de cua buida



Cas de cua no buida: `Q->start != NULL`



Stacks

Índex

1. Introducció i motivació: Aplicacions dels stacks
2. Implementació d'un stack de mida arbitrària amb una llista enllaçada
3. Exemple: càlcul de l'Span del preu d'una acció de borsa

Introducció als stacks

Un *stack* és una estructura *Last In – First Out (LIFO)* on la darrera dada en entrar és la primera en sortir:

- Un element es pot inserir en qualsevol moment a l'stack, però només el darrer l'element en entrar a l'stack pot ser retirat.
- Els elements s'insereixen i es retiren de la part superior de la pila — només es pot accedir a l'extrem anomenat la part superior de la pila.

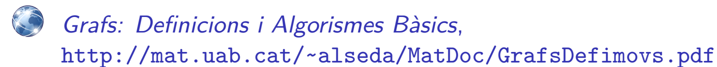
Un stack, normalment, es pensa com una estructura vertical (pila) on només es pot accedir a la part superior de la pila.

Un exemple d'aquesta estructura són les piles de plats (o de safates) per rentar d'una cafeteria.

Aplicacions dels stacks

- *Comprovació de parèntesis:* Avaluar la consistència entre l'obertura i el tancament de parèntesis.
- *Avaluació d'expressions* aritmètiques amb parèntesis.
- *Anàlisi de sintaxi:* Molts compiladors utilitzen un stack per analitzar la sintaxi d'expressions, blocs de programes, etc. abans de fer la seva traducció a codi de baix nivell.
- *Crides de funcions:* L'stack s'utilitza per guardar informació sobre quines funcions o subrutines estan actives i el seu estat.
- *Backtracking:* Recorregut de camins lògics endarrere.
- *Reversió de cadenes i fitxers*.

Una important aplicació dels stacks (que ens serveix com exemple il · lustratiu d'implementació) és la programació no recursiva de moviments per nivells en grafs.


Grafs: Definicions i Algorismes Bàsics,
http://mat.uab.cat/~alseda/MatDoc/GrafsDefimovs.pdf

Funcions fonamentals dels stacks

push Insereix un objecte a la part superior de l'stack

pop Si l'stack no està buit en retira un objecte de la part superior. En cas contrari torna un codi d'error.
La funció **pop** es pot desglossar en dues:
- **top** Llegeix la informació de l'element de la part superior de la pila.
- **deletetop** Esborra l'element de la part superior de la pila.

Així mateix un stack pot tenir associades funcions auxiliars:

StackEsBuit retorna un valor booleà que indica si l'stack està buit o no

Com en el cas de les cues, per emmagatzemar l'stack, es necessita un ***tipus de dades específic***.

La comparativa entre la programació dels stacks amb vectors de mida fixada versus llistes enllaçades de mida arbitrària coincideix totalment amb la de les cues feta a la pàgina 51. De fet, és essencialment la comparativa genèrica de vectors versus llistes enllaçades, tenint en compte les funcionalitats requerides per cues i stacks.

Implementació d'un stack amb una llista enllaçada
Declaracions i tipus de dades específics

Com sempre cal definir un tipus de dada específic que correspongui als elements individuals de la llista (stack).

Declaració dels elements de l'stack

```
typedef struct ElementDStack {
 unsigned int h;
 struct ElementDStack *lower;
} StackElem;
```

Nota: `h` és la variable destinada a contenir la informació identificativa de l'element que es posa al stack.

Com que en un stack solament volem accedir-hi per la part superior, no cal definir un *contenidor stack* complex. Podem reduir el *contenidor stack* simplement a l'apuntador que ens permeti accedir al capdamunt de la pila.

El contenidor de l'stack

```
typedef StackElem * Stack;
```

Conveni: Es fonamentar fixar i mantenir el conveni que l'stack és buit si i només si "`start`" = `NULL`.

Implementació d'un stack amb una llista enllaçada
Funcions base

Funcions senzilles de gestió de l'stack

```
int StackEsBuit( Stack St ){ return (St == NULL ); }
unsigned int top( Stack St ){ return St->h; }
```

Ús de les funcions de gestió de l'stack

Inicialització

```
Stack St = NULL;
```

La funció `top` permet recuperar o modificar dades de l'element superior de l'stack

```
htotal += top(St);
St->h = 2.0*hnou + 4;
```

Implementació d'un stack amb una llista enllaçada
Recorregut seqüencial d'una llista

StackFree: recorregut seqüencial per esborrar la llista
Calcada de la funció AlliberaCua

```
void StackFree( Stack *St ){ Stack Stindexaux = *St;
   while( Stindexaux ){ Stack top_actual = Stindexaux;
     Stindexaux = Stindexaux->lower;
     free(top_actual);
   }
   *St = NULL;
}
```

StackFree versió compacta

```
void StackFree( Stack *St ){
   while( !StackEsBuit(*St) ) deletetop(St);
   *St = NULL;
}
```

Exercici (per lliurament suplementari)
Escriure un programa que imprimeixi un fitxer amb l'ordre de línies invertit usant un stack (que contingui les línies del fitxer).

Implementació d'un stack amb una llista enllaçada
La funció deletetop
Esborrant el node inicial de l'stack

deletetop
Per poder modificar el contenidor de l'stack

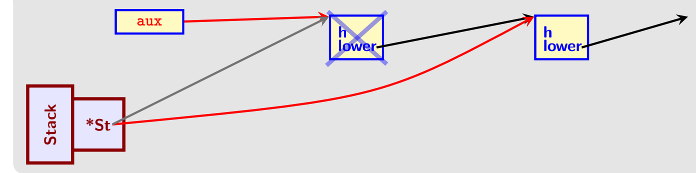
```
void deletetop( Stack *St ){
  if( StackEsBuit(*St) ) return;
  Stack aux = *St;
  *St = (*St)->lower;
  free(aux);
}
```

Quan buidem l'stack (en esborrar el darrer i únic element) reconstruïm l'assignació del conveni *St = NULL.

Exemple d'ús

```
h = top(St);
deletetop(&St);
```

deletetop en imatges



Implementació d'un stack amb una llista enllaçada
La funció push
Afegint un node al principi de la llista

push
Per poder modificar el contenidor de l'stack

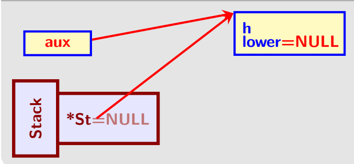
```
int push( unsigned int h, Stack *St){
  StackElem *aux =
      (StackElem *) malloc(sizeof(StackElem));
  if(aux == NULL) return 0;

  aux->h = h;

  aux->lower = *St;
  *St = aux;
  return 1;
}
```

L'assignació inicial *St = NULL assegura que lower = NULL per l'element base de la pila (és a dir el primer element que ha entrat a la pila).

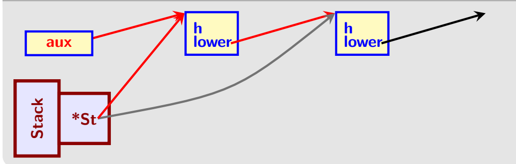
Exemple d'ús

```
if (! push(10U, &St)){
  fprintf (stderr,
    "\nERROR: Stack overfull\n"
  );
  return 11;
}
```

Cas d'stack buit



Cas d'stack no buit: *St != NULL



Exemple: càlcul de l'Span del preu d'una acció de borsa

L'*Span* del preu d'una acció de borsa en un dia donat es defineix com el nombre màxim de dies consecutius anteriors al dia donat, tals que el preu de l'acció en el dia corresponent és menor o igual que el preu en el dia donat.

Més precisament
si la sèrie de preus de l'acció en consideració és $P[0], P[1], \ldots, P[n-1]$,

$$\mathrm{Span}(i) := \max\{k \in \mathbb{Z}^+ : k \le i \quad \text{i} \quad P[j] \le P[i] \text{ per } j = i-k, i-k+1, \ldots, i\}.$$

En particular:
- $\mathrm{Span}(i) = 0 \iff P[i-1] > P[i]$ i
- $\mathrm{Span}(i) = i \iff P[i] \ge P[j]$ per $j = 0, 1, \ldots, i$.

Per tant: $\mathrm{Span}(0) = 0$.

Exemple: càlcul de l'Span del preu d'una acció de borsa

Algorisme simple — complexitat quadràtica

L'algorisme més simple per a calcular l'Span del preu d'una acció de borsa és el resultat d'aplicar directament la definició.

Algorisme simple: aplicació de la definició complexitat quadràtica (ometem la lectura de les dades)

```
#define Ndades 4448

double P[Ndades]; /* La sèrie de borsa */
unsigned int S[Ndades] = { 0U }; /* Vector d'Spans. S[0] = 0 */
register unsigned int i, k;

for (i=1; i< Ndades; i++){ S[i] = i;
    for (k=1; k<= i; k++) if (P[i-k] > P[i]) { S[i] = k-1; break; }
}
```


CC BY NC SA

Exemple: càlcul de l'Span del preu d'una acció de borsa

Algorisme amb un stack — complexitat lineal

El càlcul de l'Span del preu d'una acció de borsa es pot simplificar molt si sabem *el dia més proper abans de l'actual, denotat per $h(\cdot)$, tal que el preu de l'acció en aquest dia és més gran que el preu actual*. Si no existeix tal dia es dona el valor -1 a h. Més precisament:

Definició

$$h(i) := \begin{cases} -1 & \text{si } P[k] \le P[i] \\ & \text{per } k = 0, 1, \ldots, i-1, \\ \max\left\{k \in \mathbb{Z}^+ : i > k \text{ i } P[k] > P[i]\right\} & \text{en cas contrari.} \end{cases}$$

Observació (sobre la simplificació del càlcul d'$\text{Span}(i)$ mitjançant $h(i)$)

$$\boxed{\text{Span}(i) = i - 1 - h(i)}$$

Per aplicar aquesta simplificació al càlcul d'$\text{Span}(i)$ s'utilitza un *stack*.

CC BY NC SA

Exemple: càlcul de l'Span del preu d'una acció de borsa

Algorisme amb un stack — complexitat lineal

Definim l'*stack corresponent al dia i* com

$$\text{Stack}(i) := \left\{i > h(i) > h(h(i)) > \cdots > h^{\ell_i}(i)\right\}$$

on ℓ_i verifica que $h(h^{\ell_i}(i)) = -1$ (notem que $h^{\ell_i}(i) \ge 0$).

Clarament, $\boxed{\text{Stack}(0) = \{0\}}$

Definició: $h^0(i) := i$ i $h^\ell(i) := h\left(h^{\ell-1}(i)\right)$ per a tot $\ell \ge 1$.

Nota: Observem que l'stack mai és buit ja que conté, al menys, l'element $h^0(i) = i$ (a dalt).

Observació: L'stack pel dia i s'obté

a partir de l'stack del dia $i-1$ eliminant els elements superiors de l'stack $h^\ell(i-1)$ tals que $P\left[h^\ell(i-1)\right] \le P[i]$, i afegint $h^0(i) = i$ al damunt de la pila.

CC BY NC SA

Exemple: càlcul de l'Span del preu d'una acció de borsa

Algorisme amb un stack — complexitat lineal

Justificació

$$\boxed{\text{Stack}(i-1) := \left\{h^0(i-1) = i-1 > h(i-1) > h(h(i-1)) > \cdots > h^{\ell_{i-1}}(i-1)\right\}}$$

per a tot $i > 0$. De la definició de la funció h tenim,

$$P\left[h^{\ell_{i-1}}(i-1)\right] > P\left[h^{\ell_{i-1}-1}(i-1)\right] > \cdots > P\left[h^0(i-1)\right] = P[i-1],$$

$$P\left[h^{\ell_{i-1}}(i-1)\right] \ge P[k] \text{ per } k = 0, 1, \ldots, h^{\ell_{i-1}}(i-1), \text{ i}$$

$$P\left[h^\ell(i-1)\right] \ge P[k] \text{ per } 0 \le \ell < \ell_{i-1} \text{ i}$$
$$k = h^{\ell+1}(i-1)+1, h^{\ell+1}(i-1)+2, \ldots, h^\ell(i-1).$$

Per veure que la observació és certa, considerem dos cassos:

- $P[i] \ge P\left[h^{\ell_{i-1}}(i-1)\right] = \max\left\{P\left[h^\ell(i-1)\right] : \ell = 0, 1, \ldots, \ell_{i-1}\right\}$.
  De les desigualtats anteriors s'obté $P[i] \ge P[k]$ per $k = 0, 1, \ldots, i$.
  Per tant, $\text{Span}(i) = i$, i la observació és certa (eliminant tot $\text{Stack}(i-1)$).
- Existeix $0 \le n < \ell_{i-1}$ tal que $P\left[h^{n+1}(i-1)\right] > P[i] \ge P\left[h^n(i-1)\right]$.
  De les desigualtats anteriors per $\ell = 0, 1, \ldots, n$, s'obté $P[i] \ge P[k]$ per $k = h^{n+1}(i-1)+1, h^{n+1}(i-1)+2, \cdots, i$. Per tant,

$$h(i) = h^{n+1}(i-1) \text{ i, recursivament,}$$
$$h^2(i) = h\big(h(i)\big) = h\left(h^{n+1}(i-1)\right) = h^{n+2}(i-1),$$
$$\ldots \quad \ldots \quad \ldots$$
$$h^{\ell_{i-1}-n}(i) = h\left(h^{\ell_{i-1}-n-1}(i)\right) = h\left(h^{\ell_{i-1}-1}(i-1)\right) = h^{\ell_{i-1}}(i-1).$$

CC BY NC SA

Exemple: càlcul de l'Span del preu d'una acció de borsa

Motivació a usar el mètode de l'stack

L'algoritme directe té complexitat quadràtica Kn^2 respecte del nombre de dades n (veure el codi a la pagina 72/73) mentre que l'algorisme usant stacks té complexitat lineal $\widetilde{K}n$. Això implica una gran diferència, per n gran, com mostra la taula següent:

Nombre de dades	Alg. directe	Alg. amb stack
10^5	0"0.471'	0"0.057'
10^6	3"46.175'	0"0.589'
$2 \cdot 10^6$	18"40.350'	0"1.252'
$3 \cdot 10^6$	44"49.004'	0"1.884'
$4 \cdot 10^6$	83"44.855'	0"2.688'

Aquests resultats justifiquen amb escreix, per n gran, l'ús de l'algorisme amb stack malgrat que té una complexitat lògica superior a l'algorisme directe.

Nota

Les proves de temps de la taula anterior s'han fer amb un ordinador amb un processador Intel i7-4770K @ 3.50GHz amb 32Gb de memòria RAM DDR3 @ 1867 MHz.

Exemple: càlcul de l'Span del preu d'una acció de borsa

Implementació amb un stack de mida arbitrària amb una llista enllaçada

Algorisme usant un stack
Complexitat lineal (ometem la lectura de les dades)

```
#define Ndades 4448

double P[Ndades];
unsigned int S[Ndades] = { 0U };
register unsigned int i;
Stack St = NULL;

push(0U, &St);

for (i=1; i < Ndades ; i++){
  while( !StackEsBuit(St) && P[i] >= P[top(St)] )
                                     deletetop(&St);
  if (StackEsBuit(St)) S[i]=i; else S[i]=i-1-top(St);
  if (!push(i, &St)){ fprintf (stderr,
          "\nERROR: Stack overfull.\n\n"
      ); return 11;
    }
}
StackFree(&St);
```

La sèrie de borsa. Cal implementar la lectura de les dades des d'un fitxer.

Vector d'span's. Ja sabem que $\text{Span}(0) = 0$

Inicialització de l'stack: $\text{Stack}(0) = \{0\}$

Implementació de la observació de la pàgina 68 per a calcular $\text{Stack}(i)$ a partir de $\text{Stack}(i-1)$. Observem que, en acabar el bucle `while` de fet tenim $\text{Stack}(i) \setminus \{i\}$.

Càlcul d'$\text{Span}(i)$
És fonamental que ja s'ha calculat correctament $\text{Stack}(i)$ excepte per la no inclusió d'i. Altrament, `top(St)` retornaria $h^0(i)$ en comptes d'$h(i)$).

Finalment afegim i a $\text{Stack}(i)$; amb control d'error de manca de memòria.

Exemple: càlcul de l'Span del preu d'una acció de borsa

Implementació amb un stack de mida arbitrària amb una llista enllaçada

Declaracions i funcions base de l'stack

```
typedef struct ElementDStack {
 unsigned int h;
 struct ElementDStack *lower;
} StackElem;

typedef StackElem * Stack;

int StackEsBuit( Stack St ){ return (St == NULL ); }
unsigned int top( Stack St ){ return St->h; }
void deletetop( Stack *St ){ Stack aux = *St;
 *St = (*St)->lower;
  free(aux);
}

int push( unsigned int h, Stack *St){ StackElem *aux;
  if((aux=(StackElem *) malloc(sizeof(StackElem))) == NULL) return 0;
  aux->h=h; aux->lower=*St;
  *St = aux;
  return 1;
}

void StackFreeComp( Stack *St ){
   while( !StackEsBuit(*St) ) deletetop(St);
   *St = NULL;
}
```

L'operació `pop` l'hem dividit en dues:
`top`: retorna el valor de l'element del damunt,
`deletetop`: esborra l'element al damunt.

Com es pot veure al programa, les operacions de lectura i esborrat de l'element al damunt de la pila no sempre van juntes.

Nota: La comprovació que l'stack no és buit es fa explícitament abans de cridar aquestes funcions. No cal tornar-la a fer.