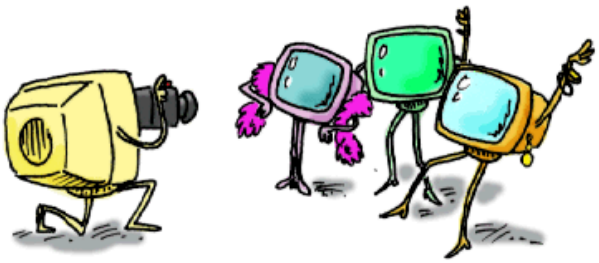
Tema 10: Estructures dinàmiques

- 10.1 Introducció
- 10.1 Els operadors new i delete
- 10.2 Introducció a les llistes vinculades
- 10.3 Assignació dinàmica
- 10.4 Operacions en llistes
- 10.5 Piles i cues
- 10.6 Arbres binaris



10.1 Introducció

- Fins ara s'ha treballat amb estructures estàtiques de dades. Això és:
  - Estructures que des d'un bon començament ja tenien un dimensionament fix.
  - Estructures que no facilitaven el poder insertar o eliminar dades d'entremig.
- Aquestes d'estructures no es poden emprar quan el contingut de la base de dades varia contínuament → En bases de dades dinàmiques.
- Fins ara, tant bon punt es definia una variable (*int[], char[],* ...) el compilador reservava l'espai de memòria adequat. Un enter ocupava 2 bytes, ...
- En estructures de dades dinàmiques, la quantitat d'elements que composaran l'array no se sap d'avançada. Per exemple, una companyia de viatges, a priori, no pot conèixer la quantitat de clients amb què comptarà.
- Una estructura de dades dinàmica permet anar ocupant espai de memòria conforme es necessita. Per tant, necessita els operadors que li permetran tractar amb més espai de memòria. Aquests operadors, de la llibreria <stdlib.h>, són:
- **L'operador malloc() en C o new() en C++**
  - Reserva el nombre de bytes que la variable necessita. Retorna l'adreça de la primera posició de memòria reservada. De no disposar de prou memòria retorna el valor NULL
- **L'operador free() en C o delete() en C++**
  - free → Allibera un bloc de bytes prèviament reservat.
- **I la funció ja coneguda sizeof()**
  - Que s'emprarà per determinar l'espai de memòra que cal reservar en cada cas.

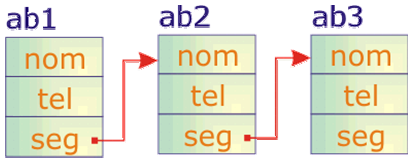
10.2 Introducció a les llistes vinculades (I)

- Les estructures de dades dinàmiques estan basades en l'ús de nodes de dades. Un node (en la seva forma elemental) consta d'una àrea de dades i una àrea d'adreça.
  - L'àrea de dades pot emmagatzemar qualsevol variable predefinida (enter, caràcter, vector, registre, ...).
  - L'àrea d'adreça conté un apuntador que conté l'adreça del node següent.

- Per exemple, la següent figura mostra una llista vinculada de tres nodes.

Pot ser constrïda a partir de tres elements d'una estructura que consta d'un nom, un número de telèfon i un apuntador al següent node.

```
struct abonat
{
        char nom[15];
        char tel[15];
        abonat *seg;
};
```



L'apuntador al següent abonat implica que, per actualitzar-se, requerirà de l'adreça d'un abonat. D'acord amb el dibuix, la inicialització de cada node serà:

```
abonat ab3 = {"Elena", " (756)012346“, NULL};   ➔ ab3.seg=NULL;
abonat ab2 = {"Pere", " (675)901234“, &ab3};    ➔ ab2.seg=&ab3;
abonat ab1 = {"Joan", " (567)890123“, &ab2};    ➔ ab1.seg=&ab2;
```


10.2 Introducció a les llistes vinculades (II)

- Com es llegeix la llista?

Qualsevol apuntador a tipus abonat serveix per ajudar a reseguir els elements de la llista. Si es fa:

```
abonat *u = &ab1;
```

Aleshores, per a accedir al nom del primer abonat ("Elena") hi ha diverses opcions:

```
ab1.nom, (*u).nom, u->nom.
```

Per apuntar al segon element hi ha les opcions directes:

```
ab2.nom, (*ab1.seg).nom, (*(*u).seg).nom, u->seg->nom
```

o, perquè no simplificar amb u = (*u).seg; (ò u = u->seg) i, aleshores

```
u->nom
```

és el nom del segon abonat, igual a "Pere"?

Es pot observar que amb l'ajut d'un apuntador a la llista és molt fàcil recòrrer la llista. Per exemple, la impressió d'aquesta llista es podria fer amb un bucle:

```
u = &ab1;
while (u != NULL)
{
        printf("Nom: %s\tTel:%s\n", u->nom, u->tel);
        u = u->seg;
}
```

10.2 Introducció a les llistes vinculades (III)

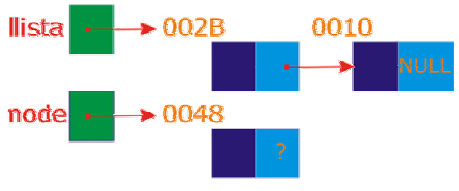
- El llistat complert de l'exemple anterior seria:

```
#include <stdio.h>
struct abonat                                   //L'estructura
{
    char nom[15];
    char tel[15];
    abonat *seg;                                //L'apuntador al següent element
};
void main()
{
    abonat ab1 = {"Joan", "(567)890123"}; //Els tres abonats
    abonat ab2 = {"Pere", "(675)901234"};
    abonat ab3 = {"Elena", "(756)012346"};
    abonat *act;                                //L'apuntador per recòrrer l'array
    ab1.seg= &ab2;                              //Lligant la llista
    ab2.seg= &ab3;
    ab3.seg= NULL;
    act = &ab1;                                 //L'element actual apunta a l'inici de la llista
    while (act != NULL)                         //Se surt del bucle quan s'apunta a l'element NULL
    {
        printf("Nom: %s\tTel:%s\n", act->nom, act->tel);
        act = act->seg;                         //Amb act recorrem la llista
    }
}
```

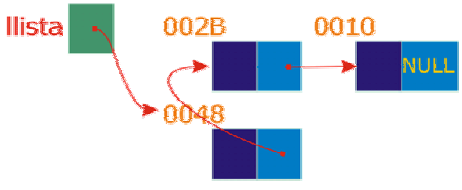

10.3 Assignació dinàmica (I)

- L'exemple anterior mostra que l'adreçament per apuntadors de llistes permet recórrer la llista, insertar i eliminar nodes sense conèixer el tamany de la base de dades. L'eina que falta introduir és la de reservar i alliberar l'espai de memòria en l'assignació dinàmica.
- En l'ús de llistes dinàmiques cal tenir present fer la reserva de memòria en el moment en què s'ha d'entrar un nou node i d'alliberar-la en el moment en què un node desapareix.
- Exemple: Es crea la mateixa estructura anterior a partir del no res. Per a fer-ho:
  - Es comença definint el primer element de la llista de forma que l'adreça següent és un element NULL. Això permetrà detectar el final de llista. Serà la cua de la llista.
  - La inserció de nodes es pot fer al començament, al final, o en un punt intermig de la llista. En aquest exemple cada node nou s'inserta al començament de la llista.
  - Per a cada node nou que s'inserta s'efectua la següent operació:

Pas 1:Donada la llista es crea un nou node. Els apuntadors llista i node apunten al començament dels seus respectius elements.



Pas 2:Amb el node creat, el camp següent del node es apunta a l'inici de la llista (node->seg = *llista;), i es canvia l'inici de la llista al nou node (*llista = node;).



10.3 Assignació dinàmica (II)

- El programa queda de la següent forma:

```
#include <stdio.h>
#include <stdlib.h>
struct abonat                   //L'estructura abonat
{
        char nom[15];
        long tel;
        abonat *seg;            //L'apuntador al següent element
};
void main()
{
        abonat *llista = NULL;  //Cap de llista
        abonat *act;            //Element de moviments
        //Entrada d'abonats
        for (int i=0; i<3; i++)
        {
            act = new abonat; //Reserva d'espai d'un nou node
            printf("--> Entra nom i numero de telefon: ");
            fflush(stdin);
            scanf("%s%i", &act->nom, &act->tel);
            act->seg = llista;      //Lligam del node amb seguent
            llista = act;           //Desplaçament del cal de llista
        }
        //Llistat de la llista
        printf("\nLa llista te els seguents nodes: \n");
        act = llista;
        while (act!=NULL)
        {
            printf("\tnom=%s tel=%i\n", act->nom, act->tel);
            act=act->seg;
        }
        //Alliberament d'espai
        while (llista!=NULL)
        {
            act=llista;
            llista=act->seg;
            delete act;             //Alliberament d'espai
        }
}
```


10.4 Operacions en llistes

- En l'actualització de bases de dades dinàmiques es fan servir una sèrie d'operacions elementals amb els seus nodes. Bàsicament són:

Inserció al començament de la llista.

- És el cas de l'exemple anterior.

Inserció al final de la llista.

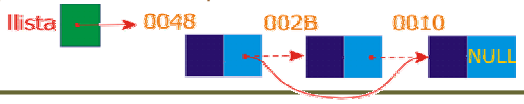
- Primer es crea el nou node i després es canvien els camps d'adreces. L'últim element apunta al nou node i el nou apunta a NULL.

Inserció enmig de la llista.

- Primer es crea el nou node i després s'inserta en la posició de l'element actual. El camp següent del nou node ha d'apuntar a on apunta el camp següent del node actual, i el camp següent del node actual al nou node.

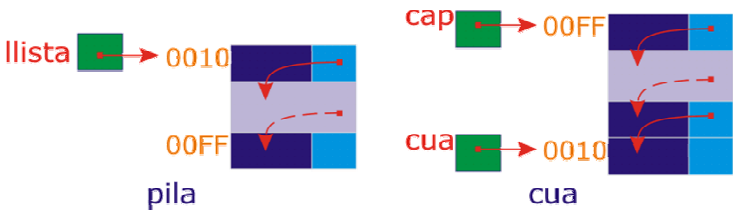
Eliminació d'un node.

- En l'eliminació del node següent a l'actual, el camp següent del node actual es fa apuntar al node de després del que s'elimina i, després, s'esborrar de memòria el node eliminat.



10.5 Piles i cues (I)

- L'ús de piles i cues controlades per software simplifica enormement la implementació d'algorismes de càlcul. El seu tractament es basa en el que s'ha vist de llistes vinculades.
- Les piles, o Last In – First Out (LIFO), són estructures de dades que permeten apilar les dades una darrera l'altra i, sempre que se'n treu alguna, l'última en entrar és la primera en sortir.
  - El tractament de les piles és similar al tractament que s'ha fet en l'exemple anterior: com que sempre es tracta amb l'últim element en entrar, tenint una variable cap de pila (en l'exemple anterior s'anomenava llista) n'hi ha prou per portar tot el control.
- Les cues són estructures First In – First Out (FIFO), en les que la primera dada en entrar és la primera en sortir.
  - En el tractament de cues fan falta dues variables per portar el control: una de capçalera i l'altre de final de cua.



10.5 Piles i cues (II)

- Un algorisme complert de tractament de dades amb una FIFO.
- També és un bon exemple per veure el pas d'estructures dinàmiques en funcions.

```
//Capçalera
//El node nou sempre s'inserta al capdavant.
Els nodes s'eliminen sempre de la cua
#include <stdio.h>
#include <stdlib.h>
struct fifo         //Estructura fifo
{
char nom[12];
fifo *seg;
};
//Declaració funció insertar node
void insertar(fifo **);
//Declaració funció eliminar node
void eliminar (fifo **);
//Procediment de llistat de la lifo
void mostrar(fifo *);
```

```
//Programa principal
void main()
{
char clau = '\0';
 fifo *cap = NULL;                  //Apuntador al cap de la LIFO
while (clau != '\n')
{
    printf("...que vols fer? (i(nsertar), e(liminar), m(ostrar): ");
    fflush(stdin);
    clau = getchar();
    switch (clau)
    {
            case 'i': insertar(&cap); //Es passa adreça on comença la llista
                    break;
            case 'e': eliminar(&cap); //Es passa adreça on comença la llista
                    break;
            case 'm': mostrar(cap); //Es passa l'element inicial
    }
}
}
```

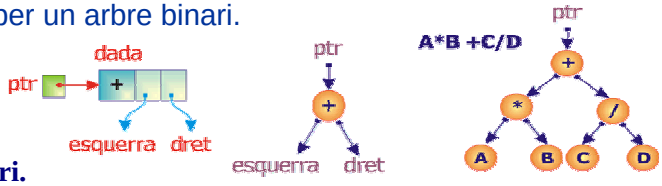
10.5 Piles i cues (III)

```
//Inserció de nodes
void insertar (fifo **capc)
{
  fifo *nde;
  nde = new fifo ;
  printf("  Entra un nom: ");
  scanf("%s", &nde->nom);
  nde->seg = *capc;
  *capc = nde;
}
//Llistar la lifo
void mostrar(fifo *ptr)
{
  while (ptr != NULL)
  {
       printf("\n%s", ptr->nom);
       ptr = ptr->seg;
  }
  printf("\n");
}
```

```
//Eliminació de nodes
void eliminar (fifo **capc)
{
  fifo *nde1, *nde2;
  if (*capc==NULL) printf( "    No hi ha cap node!\n");
  else
  {
       nde1 = *capc;
       nde2 = *capc;
       while (nde1->seg !=NULL)
       {
            nde2 = nde1;
            nde1 = nde1->seg;
       }
       nde2->seg = NULL;
       printf("     S'ha eliminat en %s\n", nde1->nom);
       delete (nde1);
       if (nde2==nde1) *capc=NULL;
  }
}
```


10.6 Arbres binaris (I)

- Els arbres binaris són estructures compostes de nodes que contenen un camp de dades i dos d'adreces, l'esquerra i el dret. Cadascun d'aquests camps d'adreces referencien, al seu temps, nodes fills situats, respectivament a sota i a l'esquerra i a la dreta del node pare. El primer node de l'arbre és el node arrel.
- Un dels usos dels arbres binaris és en l'execució d'operacions matemàtiques, com en l'exemple de la següent figura. Una fòrmula matemàtica és entrada en un arbre. Cada camp d'adreces dels nodes referencia un camp que pot contenir una dada, una operació o un element NULL. La figura de la dreta mostra com la fòrmula A*B + C/D pot estar representada per un arbre binari.



- **Recorregut d'un arbre binari.**
- Un arbre binari es comença a llegir el més avall i més a l'esquerra possible, i es va cap a la dreta. Sempre es comença a llegir quan es troba un node NULL. Un arbre binari es pot recòrrer en-ordre, en pre-ordre i en post-ordre, i dóna diferents lectures, tal com es mostra amb l'exemple de la fòrmula a*b + c/d.

En pre-ordre:	**En-ordre:**	**En post-ordre:**
Accés al camp de dades.	Accés al fill esquerra.	Accés al fill esquerra.
Accés al fill esquerra.	Accés al camp de dades.	Accés al fill dret.
Accés al fill dret.	Accés al fill dret.	Accés al camp de dades
Exem: +*AB/CD	Exem: A*B+C/D	Exem: AB*CD/+

10.6 Arbres binaris (II)

- Exemple: Treballant amb arbres.

La programació del recorregut d'un arbre binari s'implementa molt fàcilment emprant algorismes recursius, com es veu amb el següent exemple.

```
/*
L'arrel és el node 0
Per entrar l'arbre primer s'entra el nombre de nodes
Per node NULL (quan no hi ha fills) entrar 0
*/
#include <stdio.h>
#include <stdlib.h>
struct node                     //Estructura arbre
{
        char dada;
        node *esqr;
        node *dret;
};
void preordre(node *);          //Recorregut en pre-ordre
void ordre(node *);             //Recorregut en-ordre
void postordre(node *);         //Recorregut en post-ordre
node *entrar_arbre();           //Funció entrar arbre: es retorna l'adreça d'inici
```


10.6 Arbres binaris (III)

```
void main()                             //Procediment principal
{
        printf("LECTURA D'ARBRES BINARIS: \n");
        node *arrel = entrar_arbre();   //arrel és un apuntador a l'inici de l'arbre
        printf("Recorregut en pre-ordre: "); //Recorregut en pre-ordre
        preordre(arrel);
        printf("\n");
        printf("Recorregut en-ordre: ");    //Recorregut en-ordre
        ordre(arrel);
        printf("\n");
        printf("Recorregut post-ordre: ");  //Recorregut en post-ordre
        postordre(arrel);
        printf("\n");
}
void preordre(node *arrel)              //Recorregut en pre-ordre
{
        if (arrel !=NULL)
        {
                printf("%c", arrel->dada);
                preordre(arrel->esqr);
                preordre(arrel->dret);
        }
}
```

10.6 Arbres binaris (IV)

```
void ordre(node *arrel)                    //Recorregut en-ordre
{
        if (arrel !=NULL)
        {
                ordre(arrel->esqr);
                printf("%c", arrel->dada);
                ordre(arrel->dret);
        }
}
void postordre(node *arrel)                //Recorregut en post-ordre
{
        if (arrel !=NULL)
        {
                postordre(arrel->esqr);
                postordre(arrel->dret);
                printf("%c", arrel->dada);
        }
}
```


10.6 Arbres binaris (IV)

```
node *entrar_arbre()          // Funció entrar arbre
{
        int i, num, num_nod;
        printf("      Quants nodes te l'arbre? ");
        scanf("%i", &num_nod);
        node *n = new node[num_nod];
        for (i=0; i<num_nod; i++)
        {
                printf("Node [%i] \n", i);
                printf("      Dada: ");
                fflush(stdin);
                scanf("%c", &n[i].dada);
                printf("      Node esquerra: ");
                fflush(stdin);
                scanf("%i", &num);
                if (num>0) n[i].esqr = &n[num];
                else n[i].esqr = NULL;
                printf("      Node dret: ");
                fflush(stdin);
                scanf("%i", &num);
                if (num>0) n[i].dret = &n[num];
                else n[i].dret = NULL;
        }
        return (&n[0]);
}
```