Recursivitat. Arbres.

2

Isidre Guixà i Miranda

Estructures de dades dinàmiques

Maig del 2009

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`

Índex

Introducció

En la unitat didàctica precedent ens hem endinsat en la gestió dinàmica de la memòria i en la seva aplicació per a la implementació d'estructures de dades lineals: llistes, piles i cues. Ens manca, però, la seva aplicació en els arbres, un tipus de dada no lineal molt utilitzat en informàtica. Abans, però, d'introduir-nos en el coneixement dels arbres, ens cal endinsar-nos en l'aprenentatge de la recursivitat, donat que la implementació dels arbres precisen d'aquesta tècnica de programació.

Així doncs, en el nucli d'activitat "Recursivitat" introduïm aquesta tècnica de programació consistent en resoldre un problema en termes de sí mateix. El coneixement d'aquesta tècnica s'acostuma a deixar pel final de l'aprenentatge de la programació estructurada i modular donat que, per a efectuar i implementar dissenys recursius cal ser, primer de tot, un bon programador de dissenys no recursius (iteratius) i, a més, cal tenir molta intuïció per tal de saber definir la recursivitat correctament.

El fet que aquesta unitat didàctica versi sobre recursivitat i arbres podria fer pensar que els dos temes han d'anar de la mà, i no és el cas. Simplement, la implementació del tipus de dada arbre precisa de les tècniques recursives, però la recursivitat és aplicable tant al tractament del tipus de dada arbre com a moltes altres situacions en les quals els arbres no tenen res a veure.

En el nucli d'activitat "Arbres" s'introdueix aquest tipus de dada, fonamental per a la gestió dels arxius seqüencial-indexats. El tipus de dada arbre introdueix el concepte d'estructura de ramificació entre nodes. Fins ara només s'han presentat estructures lineals –estàtiques i dinàmiques– en les quals un element només té un únic següent. Amb el tipus de dada arbre s'inicia la introducció del concepte de ramificació entre nodes. És, per tant, una estructura dinàmica no lineal, molt important, però no és l'única. N'hi ha d'altres, com el graf, el qual no s'inclou en els continguts d'aquest crèdit.

D'altra banda, aquest nucli d'activitat dedicat als arbres no pretén ser una bíblia sobre els arbres. Aquest tipus de dada és tan important que s'ha estudiat llargament i, per tant, s'han generat molts conceptes. Aquí només en farem una introducció.

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 i la recursivitat són tècniques molt potents, ens cal conèixer-les i aplicar-les, però també són complexes i, per tant, cal anar amb molt de compte!

Objectius

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

1. Distingir quan un problema es pot resoldre informàticament amb la utilització de tècniques recursives.
2. Aproximar l'eficiència d'execució dels algorismes recursius dissenyats.
3. Avaluar la conveniència d'utilitzar un algorisme recursiu o un algorisme iteratiu per al disseny de determinats programes.
4. Efectuar un seguiment acurat de les diferents crides recursives que es produeixen en l'execució d'algorismes recursius.
5. Identificar diferents estructures d'arbre i implementar-les en llenguatge C.
6. Dissenyar algorismes eficients de recorregut, inserció, eliminació i consulta.
7. Aplicar les estructures d'arbre en la implementació de sistemes d'arxius seqüencials indexats.

1. Recursivitat

El concepte "recursivitat" apareix en moltes activitats de la vida diària, per exemple en una fotografia d'una fotografia, en un passadís que té miralls als extrems, encarats l'un amb l'altre, en un programa de televisió on apareix un televisor per on s'emet el mateix programa...

Però a nosaltres ens interessa el concepte "recursivitat" dins l'àmbit de la programació informàtica.

1.1. Definició i exemples

La recursivitat és aplicable en la resolució d'un problema (matemàtic, físic, informàtic,...) si la resolució del problema es pot efectuar en termes de si mateix, però en un grau de menor complexitat, 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 i, per tant, la recursivitat no deixa de ser un cas particular de crides a subprogrames.

> La recursivitat és un concepte que permet definir un objecte (problema o estructura de dades) en termes de si mateix.
>
> Un subprograma (acció o funció) és recursiu quan al llarg de la seva execució s'efectua alguna crida a si mateix.

La recursivitat en els subprogrames pot donar-se de dues maneres diferents:

- Directa, quan el subprograma es crida directament a si mateix.
- Indirecta, quan el subprograma crida un altre subprograma i aquest, o algun de més intern, crida el primer.

Davant d'una situació recursiva, ens hem de preguntar quan finalitza. El final d'un procés recursiu queda garantit amb l'existència d'un estat bàsic.

Un estat bàsic en un procés recursiu és una situació en la qual ja no s'activa més recursió.

En un subprograma recursiu, diríem que estem en un estat bàsic quan la solució al problema ja no es presenta de manera recursiva sinó directament.

A partir de la definició d'un problema que exigeix una solució informàtica, direm que una definició recursiva és vàlida si es pot determinar l'estat bàsic i es garanteix el successiu apropament del problema a l'estat bàsic esmentat.

La recursió és una eina important que permet escriure un codi de manera més compacta i, en certs casos, de manera més fàcil. Això no obstant, ha de quedar clar que la recursió no facilita més rapidesa d'execució ni tampoc permet estalviar espai de memòria. En general, és recomanable utilitzar recursió quan el problema ho exigeix. Si no és així, serà més convenient utilitzar eines de caràcter repetitiu. ‼

L'aprenentatge de l'aplicació de la recursivitat en la programació informàtica passa sempre pel coneixement d'uns exemples clàssics. En ocasions es bo plantejar-se una solució iterativa i avaluar el temps necessari per a executar la solució recursiva i la solució iterativa.

1.1.1. Càlcul del factorial d'un nombre natural

Recordem que el factorial d'un natural n es defineix com el producte dels naturals compresos entre 1 i n. Matemàticament s'expressa n!. A més, per definició, el factorial de zero és 1.

Evidentment, es pot observar que n! = n * (n - 1)!. Aquesta expressió ens defineix la recursió ja que per a calcular el factorial d'un valor ens remet al càlcul del factorial d'un altre valor. A més, observeu que el nou valor és el mateix valor decrementat en una unitat. Per tant, aplicant successivament la definició recursiva, arribaríem al valor zero, el factorial del qual és 1. Tenim, per tant, una definició recursiva del problema, que ens acosta a una situació (factorial de zero) el resultat de la qual coneixem de manera directa (estat bàsic).

La seva codificació en pseudocodi seria:

```
función factorial (n: natural) retorna natural és
   si n==0 /* estat bàsic */
```

```
      llavors retorna 1; /* final de la recursivitat */
      sinó retorna n*factorial (n-1);
/* crida recursiva que assegura l'apropament a l'estat bàsic */
   fisi
fifunció
```

La seva codificació en llenguatge C seria:

```
/* u2n1p01.c */

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

unsigned long factorial (int n)
{
   if (!n) return 1;
   return n * factorial (n-1);
}

void main()
{
   int n,aux;
   clrscr();
   do
   {  printf ("Introdueixi valor no negatiu per calcular-ne el factorial: ");
      aux=scanf("%d",&n); neteja_stdin();
   } while (aux==0 || n<0);
   printf ("\n%d! = %lu\n",n,factorial (n));
}
```


Trobareu el fitxer u2n1p01.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.

Alerta en l'execució d'aquest programa! Pensem que per l'usuari pot ser normal introduir un valor superior a 12 per a calcular-ne el factorial, no? Si agafeu una calculadora que permeti calcular el factorial d'un valor (per exemple la calculadora científica de Windows) i calculeu 12! i 13! obtindreu els resultats:

$$12! = 479001600$$
$$13! = 6227020800$$

Observem que el resultat de 13! és superior al màxim valor tractable per un `unsigned long` en plataforma de 32 bits i el programa anterior ens donaria un valor incorrecte doncs no hi controlem la superació dels valors màxims permesos. Si volem comprovar-ho, podem reversionar l'anterior programa obtenint el següent:

```
/* u2n1p02.c */

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

unsigned long factorial (int n)
```


Trobareu el fitxer u2n1p02.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.

```
/* La funció retorna ZERO si el valor és no calculable segons els tipus de dades
   emprats degut a que el valor "n" provoca resultats massa grans no tractables.
*/
{  unsigned long result;
   if (!n) return 1;
   else
   {  result = factorial (n-1);
      if (ULONG_MAX/n < result) return 0;
      else return n * result;
   }
}

void main()
{
   int n,aux;
   unsigned long resultat;
   clrscr();
   printf ("Recordem que el malor maxim per unsigned long és %lu\n",ULONG_MAX);
   do
   {  printf ("Introdueixi valor no negatiu per calcular-ne el factorial: ");
      aux=scanf("%d",&n); neteja_stdin();
   } while (aux==0 || n<0);
   if ((resultat=factorial(n))==0)
      printf ("El resultat de %d! és massa elevat i no el podem calcular.\n",n);
   else
      printf ("\n%d! = %lu\n",n,resultat);
}
```

En aquesta versió podem comprovar que l'execució de 13! ens informa que no és calculable. Fixem-nos que utilitzem la constant simbòlic `ULONG_MAX` que conté el major valor permès per a tipus `unsigned long`. Recordem, però, que en plataformes de 32 bits, algunes versions de C (*gcc*, per exemple) incorporen els tipus `long long int` i `unsigned long long int` que permeten valors més alts. Així, podem considerar una nova versió del programa:

```
/* u2n1p03.c */

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

unsigned long long factorial (int n)
/* La funció retorna ZERO si el valor és no calculable segons els tipus de dades
   emprats degut a que el valor "n" provoca resultats massa grans no tractables.
*/
{  unsigned long long result;
   if (!n) return 1;
   else
   {  result = factorial (n-1);
      if (ULLONG_MAX/n < result) return 0;
      else return n * result;
   }
}
```


Trobareu el fitxer u2n1p03.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.

```
void main()
{
   int n,aux;
   unsigned long long resultat;
   clrscr();
   printf ("Recordem que el malor maxim per unsigned long és %lu\n",ULONG_MAX);
   do
   {  printf ("Introdueixi valor no negatiu per calcular-ne el factorial: ");
      aux=scanf("%d",&n); neteja_stdin();
   } while (aux==0 || n<0);
   if ((resultat=factorial(n))==0)
      printf ("El resultat de %d! és massa elevat i no el podem calcular.\n",n);
   else
      printf ("\n%d! = %llu\n",n,resultat);
}
```

Per aconseguir la compilació en *gcc* del darrer programa cal incloure l'opció `-std=C99` en l'ordre de compilació.

1.1.2. Càlcul d'un nombre de la sèrie de Fibonacci

La sèrie de Fibonacci consisteix a obtenir, a partir dels valors 0 i 1, nous valors sumant sempre els dos anteriors:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

És clar que aquesta successió de nombres segueix la definició recursiva:

$$\text{fibonacci}(n)=\begin{cases} N & \text{si } n = 0 \text{ o } n = 1 \\ \text{fibonacci}(n-2)+\text{fibonacci}(n-1) & \text{si } n > 1 \end{cases}$$

La seva codificació en pseudocodi és:

```
funció fibonacci (n: natural) retorna natural és
   si n==0 o n==1 /* estats bàsics */
      llavors retorna n; /* final de la recursivitat */
      sinó retorna fibonacci(N-1)+fibonacci(N-2)
           /* crida recursiva amb apropament */
   fisi
fifunció
```

El següent programa en C mostra la codificació de les funcions recursives i iteratives de la funció de fibonacci i comprova la diferència de temps d'execució de les funcions recursives i iteratives.

```
/* u2n1p04.c */

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

unsigned long fibo_recursiu (unsigned long n)
{  if (n==0) return 0;
   if (n==1) return 1;
   return fibo_recursiu(n-1)+fibo_recursiu(n-2);
}

unsigned long fibo_iteratiu (unsigned int n)
{  unsigned long r1=0,r2=1,i=2,aux;
   if (n==0) return 0;
   if (n==1) return 1;
   while (i<=n) { aux=r2; r2=r2+r1; r1=aux; i++;}
   return r2;
}

void main()
{
   int n,aux;
   time_t tini,tfin;
   clrscr();
   do
   {  printf ("Introdueixi valor no negatiu per calcular-ne el seu fibonnacci: ");
      aux=scanf("%d",&n); neteja_stdin();
   } while (aux==0 || n<0);
   tini=time(NULL);
   printf ("Calculant iterativament...\n");
   printf ("ITERATIVAMENT: %lu",fibo_iteratiu(n));
   tfin=time(NULL);
   printf ("     Segons: %f\n",difftime(tfin,tini));
   tini=time(NULL);
   printf ("Calculant recursivament...\n");
   printf ("RECURSIVAMENT: %lu",fibo_recursiu(n));
   tfin=time(NULL);
   printf ("     Segons: %f\n",difftime(tfin,tini));
}
```

Trobareu el fitxer u2n1p04.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.

Executeu el programa i comproveu, per a valors relativament grans (a partir de 30 o 40), la diferència significativa de temps. Els valors obtinguts dependran, lògicament, de la velocitat de procés de la màquina on s'executin.

1.1.3. Càlcul del capital amb interès compost

Pretenem calcular el capital aconseguit després de tenir un capital `C` durant `N` anys a un interès anual `I` (percentatge) aplicant els termes del capital compost.

Sabem que el capital acumulat en finalitzar el primer any és:

```
C * (1 + I / 100)
```

Sobre aquest capital tornem a efectuar el càlcul per a obtenir el capital acumulat a la fi del segon any.

Per tant, es tracta d'un procés recursiu que podem plantejar de dues maneres, les quals queden reflectides en les dues versions en pseudocodi.

```
funció capital_compost1 (C: real, N: natural, I: real) retorna
      real és
  si N==0 /* estat bàsic */
     llavors retorna C; /* fi de la recursivitat */
     sinó retorna (1+I/100) * capital_compost1 (C, N-1, I);
                 /* crida recursiva */
  fisi
fifunció

funció capital_compost2 (C: real, N: natural, I: real) retorna
      real és
  si N==0 /* estat bàsic */
     llavors retorna C; /* fi de la recursivitat */
     sinó retorna capital_compost2 ((1+I/100)*C, N-1, I);
                 /* crida recursiva */
  fisi
fifunció
```

Vegem un programa en llenguatge C que incorpora les dues modalitats i en comprova la seva execució.

Trobareu el fitxer u2n1p05.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.

```
/* u2n1p05.c */

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

double arrodonir (double r, unsigned int n)
{  unsigned int i;
   for (i=1;i<=n;i++) r=r*10;
   if (r-floor(r)>=0.5) r = ceil(r);
   else r = floor (r);
   for (i=1;i<=n;i++) r=r/10;
   return r;
}

double capital_rec1 (double inicial, float interes, unsigned int n, unsigned int dec)
{
   if (n==0) return inicial;
   return arrodonir((1+interes/100)*capital_rec1(inicial,interes,n-1,dec),dec);
}
```

```
double capital_rec2 (double inicial, float interes, unsigned int n, unsigned int dec)
{
   if (n==0) return inicial;
   return capital_rec2(arrodonir(inicial*(1+interes/100),dec),interes,n-1,dec);
}

void main()
{
   int n,dec,aux;
   double inicial;
   float interes;
   clrscr();
   do
   {  printf ("Capital inicial (no negatiu): ");
      aux=scanf("%lf",&inicial); neteja_stdin();
   } while (aux==0 || inicial<0);
   do
   {  printf ("Número d'anys (no negatiu): ");
      aux=scanf("%d",&n); neteja_stdin();
   } while (aux==0 || n<0);
   do
   {  printf ("Interès anual (no negatiu): ");
      aux=scanf("%f",&interes); neteja_stdin();
   } while (aux==0 || interes<0);
   do
   {  printf ("Número de decimals de la moneda (no negatiu): ");
      aux=scanf("%d",&dec); neteja_stdin();
   } while (aux==0 || dec<0);
   printf ("\nVersió 1: %lf\n",capital_rec1(inicial,interes,n,dec));
   printf ("\nVersió 2: %lf\n",capital_rec2(inicial,interes,n,dec));
}
```

En aquest programa hem aplicat el càlcul real que s'efectua en les entitats bancàries (o que caldria efectuar) per a calcular el capital compost, ja que cal tenir en compte els arrodoniments de la moneda corresponent. Per això hem codificat la funció arrodonir, mitjançant la qual, donat un valor `double` i un valor `unsigned int`, s'arrodoneix el `double` al nombre de decimals que indica l'`unsigned int`. Hem de suposar que ja no tindreu cap problema per a comprendre aquesta funció. Cal utilitzar-la, posteriorment, en els càlculs dels diversos capitals acumulats.

1.2. Recursivitat i piles

Ja coneixem alguns exemples d'aplicació de la recursivitat. En general, davant la resolució informàtica d'un problema utilitzarem la recursivitat si som capaços d'observar-ne l'aplicació en el cas concret. Simplement haurem de trobar el disseny recursiu amb l'existència de l'estat base que assegura el final del problema.

Però a banda de tenir la intuïció necessària per a aplicar-la, és convenient conèixer el procés d'execució que en porta implícit la utilització. I aquest

procés es basa en la utilització continuada de piles. Anem a conèixer el funcionament intern de la recursivitat i la possibilitat de simular-lo amb la utilització de piles.

1.2.1. Funcionament intern de la recursivitat

Sabem que 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.

Així doncs, amb cada crida recursiva es crea una còpia de totes les variables i constants que estiguin vigents i es guarda a la pila. A més, es guarda una referència a la següent instrucció a executar. En retornar d'una crida recursiva, s'agafa la imatge guardada al capdamunt de la pila i es continua l'execució del subprograma en el punt on s'havia quedat en efectuar la crida recursiva. Aquesta acció es repeteix fins que la pila queda buida.

Efectuem una simulació de la recursivitat per a alguns casos.

- Càlcul de `factorial(4)`

Pas	n	Instrucció que s'executa	Pila	Factorial
0		factorial (4)		
1	4	4 * factorial (3)	4	
2	3	3 * factorial (2)	4,3	
3	2	2 * factorial (1)	4,3,2	
4	1	1 * factorial (0)	4,3,2,1	
5	0	factorial (0)	4,3,2,1	1
6		producte pendent del pas 4	4,3,2	1 (1 * 1)
7		producte pendent del pas 3	4,3	2 (2 * 1)
8		producte pendent del pas 2	4	6 (3 * 2)
9		producte pendent del pas 1		24 (4 * 6)

- Càlcul de `fibonacci(4)`

Pas	Instrucció que s'executa	Observacions	Estat de la pila (de baix cap a dalt)
0	`fibo(4)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 1.	
1	`retorna fibo(3)+fibo(2)`	Per poder executar el **retorna** cal, prèviament, calcular l'expressió que acompanya el **retorna**, en la qual hi ha dues crides a funcions. Per tant, primer el processador procedirà a processar la primera crida `fibo(3)` i la resta queda pendent de processar, a la pila.	
2.	`fibo(3)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 3.	`retorna ResultatPas2+fibo(2)`
3	`retorna fibo(2)+fibo(1)`	Per poder executar el **retorna** cal, prèviament, calcular l'expressió que acompanya el **retorna**, en la qual hi ha dues crides a funcions. Per tant, primer el processador procedirà a processar la primera crida `fibo(2)` i la resta queda pendent de processar, a la pila.	`retorna ResultatPas2+fibo(2)`
4	`fibo(2)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 5.	`retorna ResultatPas4+fibo(1)` `retorna ResultatPas2+fibo(2)`
5	`retorna fibo(1)+fibo(0)`	Per poder executar el **retorna** cal, prèviament, calcular l'expressió que acompanya el **retorna**, en la qual hi ha dues crides a funcions. Per tant, primer el processador procedirà a processar la primera crida `fibo(1)` i la resta queda pendent de processar, a la pila.	`retorna ResultatPas4+fibo(1)` `retorna ResultatPas2+fibo(2)`

6	`fibo(1)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 7.	`retorna ResultatPas6+fibo(0)`
			`retorna ResultatPas4+fibo(1)`
			`retorna ResultatPas2+fibo(2)`

7	`retorna 1`	En aquesta crida s'obté un resultat concret (1) i finalitza. El processador recupera del capdamunt de la pila el què té pendent de fer i sap que el valor 1 obtingut correspon a `ResultatPas6`. La pila disminueix.	`retorna ResultatPas6+fibo(0)`
			`retorna ResultatPas4+fibo(1)`
			`retorna ResultatPas2+fibo(2)`

8	`retorna 1+fibo(0)`	Per poder executar el **retorna** cal, prèviament, calcular l'expressió que acompanya el **retorna**, en la qual hi ha una crida a funció. Per tant, primer el processador procedirà a processar la crida `fibo(0)` i la resta queda pendent de processar, a la pila.	`retorna ResultatPas4+fibo(1)` `retorna ResultatPas2+fibo(2)`

9	`fibo(0)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 10.	`retorna 1+ResultatPas9`
			`retorna ResultatPas4+fibo(1)`
			`retorna ResultatPas2+fibo(2)`

10	`retorna 0`	En aquesta crida s'obté un resultat concret (0) i finalitza. El processador recupera del capdamunt de la pila el què té pendent de fer i sap que el valor 0 obtingut correspon a `ResultatPas9`. La pila disminueix.	`retorna 1+ResultatPas9`
			`retorna ResultatPas4+fibo(1)`
			`retorna ResultatPas2+fibo(2)`

11	`retorna 1+0`	En aquesta expressió obté un resultat concret (1) i finalitza. El processador recupera del capdamunt de la pila el què té pendent de fer i sap que el valor 1 obtingut correspon a `ResultatPas4`. La pila disminueix.	`retorna ResultatPas4+fibo(1)` `retorna ResultatPas2+fibo(2)`
12	`retorna 1+fibo(1)`	Per poder executar el **retorna** cal, prèviament, calcular l'expressió que acompanya el **retorna**, en la qual hi ha una crida a funció. Per tant, primer el processador procedirà a processar la crida `fibo(1)` i la resta queda pendent de processar, a la pila.	`retorna ResultatPas2+fibo(2)`
13	`fibo(1)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 14.	`retorna 1+ResultatPas13` `retorna ResultatPas2+fibo(2)`
14	`retorna 1`	En aquesta crida s'obté un resultat concret (1) i finalitza. El processador recupera del capdamunt de la pila el què té pendent de fer i sap que el valor 1 obtingut correspon a `ResultatPas13`. La pila disminueix.	`retorna 1+ResultatPas13` `retorna ResultatPas2+fibo(2)`

Pas	Instrucció que s'executa	Observacions	Estat de la pila (de baix cap a dalt)
15	`retorna 1+1`	En aquesta expressió obté un resultat concret (2) i finalitza. El processador recupera del capdamunt de la pila el què té pendent de fer i sap que el valor 2 obtingut correspon a `ResultatPas2`. La pila disminueix (quedarà buida)	`retorna ResultatPas2+fibo(2)`
16	`retorna 2+fibo(2)`	Per poder executar el **retorna** cal, prèviament, calcular l'expressió que acompanya el **retorna**, en la qual hi ha una crida a funció. Per tant, primer el processador procedirà a processar la crida `fibo(2)` i la resta queda pendent de processar, a la pila.	
17	`fibo(2)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 18.	`retorna 2+ResultatPas17`
18	`retorna fibo(1)+fibo(0)`	Per poder executar el **retorna** cal, prèviament, calcular l'expressió que acompanya el **retorna**, en la qual hi ha dues crides a funcions. Per tant, primer el processador procedirà a processar la primera crida `fibo(1)` i la resta queda pendent de processar, a la pila.	`retorna 2+ResultatPas17`
19	`fibo(1)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 20.	`retorna ResultatPas19+fibo(0)` `retorna 2+ResultatPas17`
20	`retorna 1`	En aquesta crida s'obté un resultat concret (1) i finalitza. El processador recupera del capdamunt de la pila el què té pendent de fer i sap que el valor 1 obtingut correspon a `ResultatPas19`. La pila disminueix.	`retorna ResultatPas19+fibo(0)` `retorna 2+ResultatPas17`
21	`retorna 1+fibo(0)`	Per poder executar el **retorna** cal, prèviament, calcular l'expressió que acompanya el **retorna**, en la qual hi ha una crida a funció. Per tant, primer el processador procedirà a processar la crida `fibo(0)` i la resta queda pendent de processar, a la pila.	`retorna 2+ResultatPas17`
22	`fibo(0)`	El processador inicia l'execució de la crida i obté que ha d'efectuar el càlcul indicat en el pas 23.	`retorna 1+ResultatPas22` `retorna 2+ResultatPas17`
23	`retorna 0`	En aquesta crida s'obté un resultat concret (0) i finalitza. El processador recupera del capdamunt de la pila el què té pendent de fer i sap que el valor 0 obtingut correspon a `ResultatPas22`. La pila disminueix.	`retorna 1+ResultatPas22` `retorna 2+ResultatPas17`
24	`retorna 1+0`	En aquesta expressió obté un resultat concret (1) i finalitza. El processador recupera del capdamunt de la pila el què té pendent de fer i sap que el valor 1 obtingut correspon a `ResultatPas17`. La pila disminueix (quedarà buida)	`retorna 2+ResultatPas17`
25	`retorna 2+1`	En aquesta expressió obté un resultat concret (3) i finalitza. Com que la pila ha quedat buida, vol dir que és el resultat final del càlcul inicial: `fibo(4)`	

Enteneu per què la versió recursiva Fibonacci és molt ineficient? Fixeu-vos que van quedant un munt de crides pendents d'executar a la pila. Quan arriba el moment de la seva execució, s'executen. Algunes ja s'havien calculat abans, però l'ordinador no ho pot saber i les torna a executar. ¿Enteneu que l'ordinador pot quedar "penjat" a causa de la saturació de la pila? En aquest cas, la versió recursiva hauria de quedar totalment descartada.

1.2.2. Simulació de la recursivitat amb piles

No tots els llenguatges de programació disposen de recursivitat. D'altra banda, ja hem comentat més amunt que la utilització de la recursivitat no és prou eficient. Resulta convenient, per tant, comptar amb algun

procediment que permeti simular la recursivitat. I com que ja som coneixedors del tipus de dada pila, podem utilitzar-la gestionant nosaltres mateixos la memòria disponible.

Un subprograma pot tenir variables locals i paràmetres (arguments). Quan es fa una crida recursiva al subprograma, els valors actuals de variables i paràmetres han de conservar-se. També ha de conservar-se l'adreça a la qual ha de tornar el control una vegada executat el subprograma cridat. En realitat, aquest procediment és el que s'utilitza sempre que es fa la crida d'un subprograma a un altre (encara que no hi hagi recursivitat).

El procediment que es proposa per a simular la recursivitat d'un algorisme que tingui `N` paràmetres i `M` variables locals consisteix a tenir `N+M` piles (o una pila de `N+M` valors per element) on es puguin guardar les imatges corresponents del moment en què cal fer la crida i refer la recursivitat en un procés iteratiu.

Utilitzarem les piles amb les dues operacions ja conegudes:

```
funció push (var top: ^node, elem: T) retorna natural;
/* 0: O.K. - 1: Pila plena */

funció pop (var top: ^node, var elem: T) retorna natural;
/* 0: O.K. - 1: Pila buida */
```

Solucionarem ara el càlcul del factorial d'un nombre natural simulant la recursivitat.

```
funció factorial (n: natural) retorna natural és
   var top: ^natural; r: natural; aux: natural; fivar
   top = NUL; /* Punter al capdamunt de la pila */
   /* El següent mentre equival a anar guardant el valor n de manera que quan es tingui
      r=factorial(n-1) es pugui recuperar per a calcular n*r */
   mentre n > 0 fer
      si push (top, n) != 0 llavors ERROR("Pila plena"); avortar; fisi
      n = n - 1;
   fimentre
   r = 1; /* Solució per l'estat base */
   /* El següent mentre equival a tornar de les diferents crides recursives recuperant
      els valors de la pila i fent els càlculs oportuns */
   mentre pop (top, aux) == 0 fer r = r * aux; fimentre
   retorna r;
fifunció
```

En veure-ho d'aquesta manera ens adonem que podem estalviar-nos l'emmagatzematge a la pila del valor `n`, ja que consisteix a anar guardant els diferents valors des de `n` fins a 1 (descendentment), recuperar-los posteriorment d'un en un i anar calculant `r`. En aquest cas tindríem la solució iterativa coneguda per tothom:

```
funció factorial (n: natural) retorna natural és
   var r: natural; aux: natural; fivar
   r=1; aux=1;
   mentre aux <= n fer r = r*aux; aux = aux+1; fimentre
   retorna r;
fifunció
```

1.3. Aplicacions clàssiques de recursivitat

Ja ens són coneguts els típics exemples senzills d'utilització de recursivitat: factorial, Fibonacci i capital compost. Hi ha, però, dos exemples clàssics d'aplicació de recursivitat. El primer és la resolució d'un problema (torres de Hanoi) en què es pot apreciar que el disseny recursiu el simplifica increïblement en comparació d'un disseny iteratiu. El segon és el mètode d'ordenació ràpida (quicksort), al qual s'ha al·ludit en moltes ocasions.

1.3.1. El problema de les torres de Hanoi

Aquest és un clàssic de la recursivitat ja que la seva solució recursiva és molt simple. Es tenen tres torres (`T1`, `T2` i `T3`) i un conjunt de `N` discos de diferents diàmetres. Cadascun té una perforació central que li permet lliscar per qualsevol de les torres. Inicialment (figura 1), els `N` discos estan situats a la torre `T1`, ordenats per diàmetre, de més gran (a sota) a més petit (a dalt). S'han de passar els `N` discos a la torre `T2` (figura 2), on han de quedar en el mateix ordre), utilitzant la torre `T3` com a auxiliar i respectant les regles següents:

Figura 1. Situació inicial dels N discos en el problema de les Torres de Hanoi

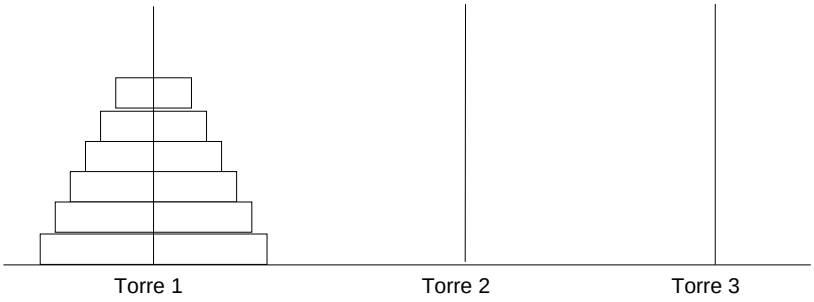
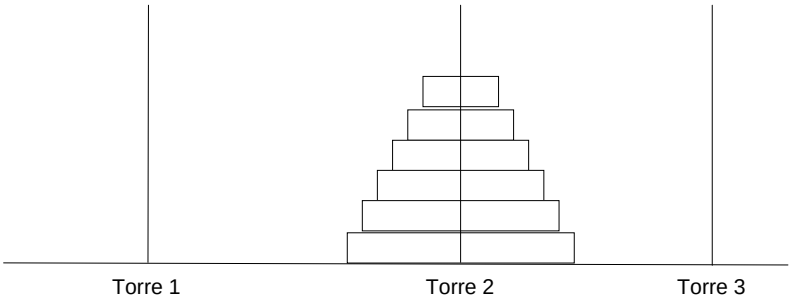


Figura 2. Situació final dels N discos en el problema de les Torres de Hanoi



- A cada moviment només hi pot intervenir un disc; per tant, sempre serà el disc superior el que es pugui moure.

- No pot quedar en cap moment un disc damunt d'un altre que tingui menys diàmetre.

L'única manera de passar el disc inferior de `T1` a `T2` respectant les regles és:

1) Movent els `N-1` discos superiors de `T1` a `T3`, utilitzant la torre T2 com auxiliar.

2) Movent el disc inferior de `T1` a `T2`.

3) Movent els `N-1` discos existents de `T3` a `T2`, utilitzant la torre T1 com auxiliar.

Observeu que si els moviments dels apartats 1) i 3) respecten les regles (considerant només els `N-1` discos que mouen), llavors la totalitat dels moviments 1)-2)-3) també les respecten, ja que els discos que es mouen sempre són de menys diàmetre que els que ja estan situats a les torres. Per tant, s'ha aconseguit redefinir el problema de grau `N` en termes de grau `N-1`. Ens estem apropant a un estat base (un únic disc), on el moviment entre la torre font i la torre destí és immediat, és a dir, sense l'ajut de la torre auxiliar.

```
acció hanoi (N, FONT, DESTÍ, AUXILIAR: natural) és
   si N==1
      llavors escriure ("Posar disc de ",FONT," a ", DESTÍ);
              /* Estat bàsic */
      sinó hanoi (N-1, FONT, AUXILIAR, DESTÍ);
           escriure ("Posar disc de ",FONT," a ", DESTÍ);
           hanoi (N-1, AUXILIAR, DESTÍ, FONT);
   fisi
fiacció
```

Presentem, a continuació, una versió no recursiva utilitzant piles per a simular la recursió.

```
tipus sit_hanoi = tupla
                    N, TF, TD, TA: natural;
                  fitupla
    /* El tipus sit_hanoi permet emmagatzemar una situació concreta del problema */
    node = tupla
             inf: sit_hanoi;
             seg: ^node;
           fitupla
    /* El tipus node correspon al node per a una pila implementada com a llista */
fitipus
```

```
acció hanoi (N, FONT, DESTÍ, AUXILIAR: natural) és
  var top: ^node; sit: sit_hanoi; aux, crida: natural; fivar
  crida=1;
  mentre N>1 i crida==1 fer
  /* El següent mentre posa a la pila totes les primeres crides */
    mentre N > 1 fer
       sit.N=N; sit.TF=FONT; sit.TD=DESTÍ; sit.TA=AUXILIAR;
       si push (top, sit)!=0 llavors ERROR("Pila plena"); avortar; fisi
       N=N-1; aux=DESTÍ; DESTÍ=AUXILIAR; AUXILIAR=aux;
    fimentre
    escriure ("Posar disc de ",FONT," a ", DESTÍ); crida=2;
    si pop(p,sit)==0 llavors
       N=sit.N; FONT=sit.TF; DESTÍ=sit.TD; AUXILIAR=sit.TA;
       escriure ("Posar disc de ",FONT," a ", DESTÍ);
       /* Simularem la 2a crida a hanoi corresponent a la que acabem de tractar */
       N=N-1; aux=FONT; FONT=AUXILIAR; AUXILIAR=aux; crida=1;
    fisi
  fimentre
fiacció
```

Una simple ullada entre els algorismes recursiu i iteratiu permet concloure que la solució recursiva del problema de les torres de Hanoi és molt més compacta i més fàcil d'entendre que la solució no recursiva. De tota manera, és important disposar d'un formalisme per a traduir els processos recursius en processos no recursius, ja que alguns llenguatges no disposen de recursivitat i en certs problemes el cost de la recursivitat és molt significatiu.

1.3.2. El mètode d'ordenació ràpida

El mètode d'ordenació ràpida, batejat d'aquesta manera pel seu dissenyador (Hoare) i també conegut com a mètode d'ordenació per partició, és un mètode d'ordenació de taules molt millor que els ja coneguts mètodes de selecció, inserció i intercanvi i molts llenguatges el faciliten en les seves llibreries de funcions.

Aquest mètode d'ordenació té un disseny recursiu molt fàcil, a partir del qual i amb la utilització de piles, es pot aconseguir una versió iterativa.

L'algorisme consisteix, fonamentalment, en els passos següents:

1) S'agafa un element `X` d'una posició qualsevol de la taula.

2) S'intenta col·locar `X` a la posició correcta de la taula, de tal manera que tots els elements que es trobin a la seva esquerra siguin més petits o iguals que `X` i que tots els elements que es trobin a la seva dreta siguin més grans o iguals que `X`. Per aconseguir-ho, caldrà, probablement, moure diversos elements de la taula.

3) Es repeteixen els passos anteriors, però ara per als conjunts de dades que es troben a l'esquerra i a la dreta de la posició correcta de X dins la taula.

4) El procés acaba quan tots els elements es troben en la posició correcta dins la taula.

Del que s'ha exposat es pot deduir que aquest algorisme és fàcilment programable si s'utilitza la recursió. Vegem-ne la versió recursiva.

Seleccionarem un element X qualsevol. Considerarem, per exemple, t[0]. Comencem a recórrer la taula de dreta a esquerra comparant si els elements són més grans o iguals que X. Si un element no compleix la condició, s'intercanvia amb X. S'inicia novament el recorregut a partir d'on ara està col·locat X, però d'esquerra a dreta, comparant si els elements són més petits o iguals que X. Si un element no compleix la condició, s'intercanvia amb X. Es repeteixen els passos anteriors (a partir d'on ara està col·locat X) fins que l'element X troba la seva posició correcta dins la taula.

Per ordenar una taula t[MAX] de T, caldria executar quicksort (0,MAX-1) on:

```
acció quicksort (inici, final: natural) és
   var esq, dre, pos: natural; situat: booleà; aux: T; fivar
   esq = inici; dre = final; pos = inici; situat = fals;
   mentre no situat fer
      situat = cert;
      mentre t[dre].V >= t[pos].V i dre != pos fer dre = dre -1; fimentre
      si dre != pos llavors
         aux = t[pos]; t[pos] = t[dre]; t[dre] = aux; pos = dre;
         mentre t[esq].V <= t[pos].V i esq != pos fer esq = esq + 1; fimentre
         si esq != pos llavors
            situat = fals; aux = t[pos]; t[pos] = t[esq]; t[esq] = aux; pos = esq;
         fisi
      fisi
   fimentre
   /* En aquest moment l'element que es troba a la posició "pos" té tots els elements
      de la seva esquerra menors o iguals que ell i tots els elements de la seva dreta
      majors o iguals que ell.
   */
   si inici < pos - 1 llavors quicksort (inici, pos - 1); fisi
   si final > pos + 1 llavors quicksort (pos + 1, final); fisi
fiacció
```

Càlculs matemàtics asseguren que aquest algorisme és d'ordre $\theta(n * \log_2 n.)$.

Encara que l'algorisme del quicksort en versió recursiva sigui clar, és possible augmentar-ne l'eficiència eliminant les crides recursives. La recursió és un instrument molt poderós, però l'eficiència d'execució és

un factor molt important en un procés d'ordenació que cal administrar molt bé. Aquestes crides recursives poden substituir-se utilitzant piles, cosa que dóna lloc a la versió iterativa.

Us atreviu a dissenyar-ne la versió iterativa?

2. Arbres

Un arbre és una estructura jeràrquica aplicada sobre una col·lecció d'elements, anomenats nodes, un dels quals és conegut com a arrel. Entre els diferents nodes es crea una relació o parentesc, que dóna lloc a termes com pare, fill, germà...

> Un arbre de tipus T és una estructura homogènia fruit de la concatenació d'un element de tipus T, anomenat arrel, amb un nombre finit d'arbres disjunts (que no tenen cap element en comú), anomenats subarbres. Un cas particular d'arbre pot ser l'estructura buida.

Observeu que la definició d'arbre ha estat recursiva, ja que per definir un arbre s'ha utilitzat el mateix concepte d'arbre. L'última part de la definició (un arbre pot ser buit) constitueix l'estat base de la definició recursiva que n'assegura la finitud.

Els arbres tenen una gran varietat d'aplicacions: representació de fórmules matemàtiques, representació d'arbres genealògics, anàlisi de circuits elèctrics, esquemes de continguts... Però, sobretot, s'utilitzen en la gestió dels índexs en els arxius seqüencials indexats.

Les característiques i propietats dels arbres en general són: a

a) Tot arbre no buit té un únic node arrel.

b) Un node X és descendent directe d'un node Y, si el node X és apuntat pel node Y. En aquest cas s'acostuma a utilitzar l'expressió X és fill de Y.

c) Un node X és ascendent directe d'un node Y, si el node X apunta el node Y. En aquest cas s'acostuma a utilitzar l'expressió X és pare de Y.

d) Tots els nodes que són fills d'un mateix node s'anomenen germans.

e) Tot node sense fills es coneix com a node terminal o fulla.

f) Tot node que no és arrel ni fulla es coneix com a node interior.

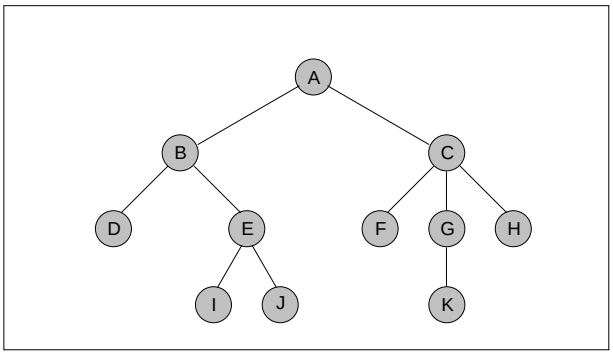
g) El grau d'un node es defineix com el nombre de fills del node, i el grau d'un arbre es defineix com el grau més gran de tots els nodes de l'arbre.

h) El nivell d'un node es defineix com el nombre de generacions entre l'arrel i l'esmentat node. Per definició, el node arrel és de nivell 1.

i) L'altura d'un arbre es defineix com el nivell més gran de tots els nodes de l'arbre.

Vegem aquestes propietats a partir de l'arbre de la figura 3:

Figura 3. Exemple d'arbre genèric



- A és l'arrel.
- D, I, J, F, K i H són nodes terminals fulles.
- B, C, E i G són nodes interiors.
- E és pare de I i J, i aquests són fills de E.
- G és pare de K, i aquest és fill de G.
- B és pare de D i E, i aquests són fills de B.
- C és pare de F, G i H, i aquests són fills de C.
- A és pare de B i C, i aquests són fills de A.
- El grau de B és 2.
- El grau de l'arbre és 3, ja que C té grau 3 i és el grau més gran.
- L'altura de l'arbre és 4, ja que és el nivell més alt (per a les fulles I, J i K).

A partir de la figura 3 i de la nostra experiència prèvia intuïm que informatitzar una estructura d'aquest estil implicarà la utilització de nodes en els quals emmagatzemarem les dades (A, B, C, D...) i els punters, tants com fills tingui aquell node. No informatitzarem, però, qualsevol tipus d'arbre sinó que anirem filtrant els tipus d'arbres interessants per al tractament informàtic.

2.1. Arbres binaris

Els primers arbres que tenen una certa importància i que ens interessa tractar informàticament són els arbres binaris.

Un arbre ordenat és un arbre les branques dels nodes del qual estan ordenades o, millor dit, es poden enumerar.

Un arbre binari és un arbre ordenat de grau 2. Per tant, cada node pot tenir, com a màxim, dos fills. Quan s'ordena un arbre binari sempre es pot distingir entre el subarbre esquerre (arbre que té com a arrel el fill esquerre) i el subarbre dret (arbre que té com a arrel el fill dret).

Un arbre binari és complet si tots els seus nodes que no siguin fulles tenen dos fills i totes les fulles es troben en el mateix nivell.

En un arbre complet es verificarà que:

$$\text{Nombre de nodes} = 2^h - 1 \quad (h \text{ és l'altura de l'arbre})$$

La importància dels arbres binaris radica en el seu dinamisme, la seva senzilla programació, la no linealitat i el fet que existeixen algorismes mitjançant els quals qualsevol estructura en arbre pot ser reduïda a un arbre binari i, per tant, tenir el tractament informàtic corresponent a un arbre binari.

Conversió a arbre binari

No s'inclou en els objectius d'aquest crèdit la mecànica de transformació de qualsevol arbre no binari en binari. En els llibres de la bibliografia específica corresponents a aquest aspecte hi trobareu les explicacions.

2.1.1. Implementació d'arbres binaris

De la mateixa manera que en el tipus de dada pila i cua existeix la possibilitat de fer una implementació basada en taules, encara que és millor que es basi en llistes, en els arbres binaris ens passa quelcom semblant. És possible una implementació basada en taules, però no és gens lògica. Així doncs, la normal implementació d'arbres passa per la gestió dinàmica de memòria.

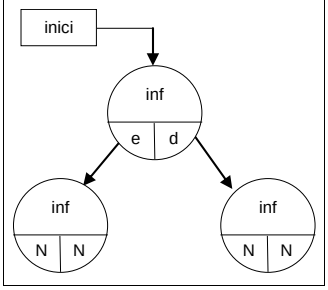
Un node d'un arbre binari consta de tres parts:

- Una part que conté la informació que es vol emmagatzemar a l'arbre.
- Una part que conté el lligam cap al fill esquerre. Contindrà el valor NUL si no té fill esquerre.
- Una part que conté el lligam cap al fill dret. Contindrà el valor NUL si no té fill dret.

Per tant, si volem gestionar un arbre per a emmagatzemar-hi dades d'un cert tipus T, caldrà tenir en compte la definició següent:

```
tipus node = tupla
            inf : T; /* conté la informació */
            esq, dre : ^node; /* punters als dos fills */
          fitupla
```

Figura 4. Utilització de punters en nodes dels arbres binaris



La figura 4 ens ho mostra gràficament. És clar, per tant, que:

- un arbre binari és una estructura ramificada de nodes per a la qual hi ha d'haver un punter inicial,
- les fulles tenen el valor NULL (N) en els dos punters,
- un arbre buit té en el punter inicial el valor NULL.

2.1.2. Algorismes de tractament en arbres binaris

Analitzem, a continuació, els principals algorismes de tractament en arbres binaris: creació d'arbre buit, càrrega i recorreguts.

Podríem estudiar altres algorismes, però ens interessen, sobretot, els recorreguts que es poden aplicar. Per a poder-los comprovar necessitem tenir algun arbre amb informació introduïda. Per això considerem, també, un algorisme de càrrega d'un arbre binari a partir d'un conjunt de valors inicial.

1) Creació d'arbre buit

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

Evidentment, crear l'arbre no consisteix a fer res més que crear el punter inicial. Cal tenir, però, la precaució d'inicialitzar-lo amb el valor NUL.

2) Càrrega d'un arbre binari

No ens complicarem la vida. Simplement volem disposar d'algun mètode per a carregar en memòria un arbre binari que tinguem en paper per tal de poder comprovar, posteriorment, els algorismes de recorregut, que són el motiu fonamental d'aquest apartat.

```
acció càrrega (var arbre:^node) retorna natural és
/* Argument arbre : Punter per a apuntar l'arrel de l'arbre */
   var element: T; sn: caràcter; fivar
   si arbre != NUL llavors esborrar (arbre); fisi
   escriure ("Introdueixi l'element a inserir:"); llegir (element);
```

```
   assignar_memòria (arbre);
   si arbre == NUL llavors escriure ("L'arbre no pot créixer més.");
   sinó arbre^.inf = element; arbre^.esq = NUL; arbre^.dre = NUL;
        pregunta ("Té fill per l'esquerra?", sn, "SN");
        si sn == 'S' llavors
           càrrega (arbre^.esq); fisi /* crida recursiva */
           escriure ("Situïs en el node que té com a element", element);
           /* Pregunta perquè l'usuari se situï en cas d'haver-se executat
              la crida recursiva i haver deixat "penjat" el node dret d'aquest
              element */
        fisi
        pregunta ("Té fill per la dreta?", sn, "SN");
        si sn == 'S' llavors càrrega (arbre^.dre); fisi /* crida recursiva */
   fisi
fiacció
```

3) Algorismes de recorregut

Aquests són els algorismes més importants en el tractament d'arbres binaris en general, ja que recórrer un arbre significa visitar-ne els nodes sistemàticament, de manera que tots siguin visitats una única vegada.

Cal entendre com a visita el fet de tractar la informació que conté. No es considera visita el fet de passar pel node per a consultar els punters `esq` i `dre` i poder passar així als nodes fills corresponents.

Hi ha tres maneres de fer el recorregut d'arbres binaris. Totes són de naturalesa recursiva: preordre, inordre i postordre.

a) El recorregut en preordre consisteix a visitar, en primer lloc, l'arrel, en segon lloc, el subarbre esquerre en preordre i, en tercer lloc, el subarbre dret en preordre.

b) El recorregut en inordre consisteix a visitar, en primer lloc, el subarbre esquerre en inordre, en segon lloc, l'arrel i, en tercer lloc, el subarbre dret en inordre.

c) El recorregut en postordre consisteix a visitar, en primer lloc, el subarbre esquerre en postordre, en segon lloc, el subarbre dret en postordre i, en tercer lloc, l'arrel.

Observeu que les tres definicions són recursives. La codificació dels tres recorreguts serà, per tant, recursiva.

```
acció preordre (arbre:^node) és
   si arbre != NUL llavors
      tractament (arbre^.inf);
      preordre (arbre^.esq);
```

```
      preordre (arbre^.dre);
   fisi
fiacció

acció inordre (arbre:^node) és
   si arbre != NUL llavors
      inordre (arbre^.esq);
      tractament (arbre^.inf);
      inordre (arbre^.dre);
   fisi
fiacció

acció postordre (arbre:^node) és
   si arbre != NUL llavors
      postordre (arbre^.esq);
      postordre (arbre^.dre);
      tractament (arbre^.inf);
   fisi
fiacció
```

2.2. Arbres binaris de recerca

Els arbres binaris són molt interessants, però encara ho serien més si disposéssim de mètodes de recerca eficients. Com cercaríeu un element en un arbre binari genèric de l'apartat anterior? Només hi ha una manera de fer-ho: considerar un dels tres recorreguts possibles i anar cercant fins a trobar l'element o fins a arribar al final del recorregut sense èxit. Gens eficient!

Aquest algorisme és tan ineficient com ho era la recerca seqüencial en taules. Recordeu com varem poder millorar el cost lineal de la recerca seqüencial? Ho varem aconseguir en les taules ordenades, ja que llavors es podia aplicar la recerca dicotòmica.

En el tipus de dada arbre succeeix una cosa semblant i per aquest motiu interessa treballar amb arbres binaris de recerca.

Un arbre binari de recerca és un arbre binari tal que:

- Tots els valors emmagatzemats en el subarbre esquerre de qualsevol node T han de ser més petits o iguals que el valor emmagatzemat en el node T.
- Tots els valors emmagatzemats en el subarbre dret de qualsevol node T han de ser més grans que el valor emmagatzemat en el node T.

Intuïu per on anirà la recerca d'un determinat element? Donat un node, si l'element cercat té un valor més petit que el valor emmagatzemat en el node, continuarem la recerca pel subarbre esquerre, i si el seu valor és més gran, seguirem la recerca pel subarbre dret. Evidentment, si el valor coincideix amb el valor del node, aturarem la recerca amb un resultat positiu. Fixeu-vos que és una recerca dicotòmica en arbres, ja que a cada node obviem, mitjançant el procés de recerca, un dels dos subarbres.

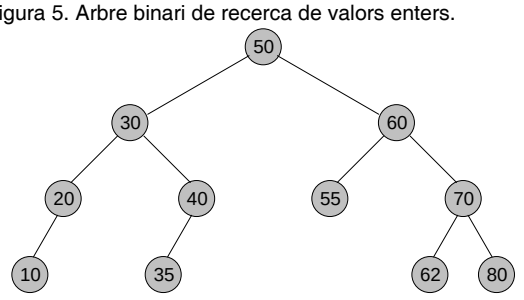
Comparació amb altres tipus de dades

Observeu que en les taules ordenades la recerca és eficient, però mantenir-la ordenada és costós. En les llistes ordenades són costosos el manteniment i la recerca.

Aquests arbres no solament tenen importància gràcies al seu algorisme de recerca. Permeten, a més, realitzar de manera eficient les operacions d'inserció i eliminació de manera que l'arbre mantingui sempre la categoria d'arbre de recerca.

La figura 5 ens mostra un arbre binari de recerca. Observeu que un recorregut en inordre provoca una classificació ascendent de tots els seus nodes:

Figura 5. Arbre binari de recerca de valors enters.



10 - 20 - 30 - 35 - 40 - 50 - 55 - 60 - 62 - 70 - 80

Vegem els algorismes de recerca, inserció i esborrament en arbres binaris de recerca, suposant que no hi ha elements repetits (cas més freqüent).

2.2.1. Algorisme de recerca

L'algorisme de recerca consisteix en el fet que a cada node pel qual es passa es produeixi una dicotomia.

```
funció recerca (arbre:^node, var elem:T, ref: T*) retorna natural és
/* Arguments:
      arbre: Punter a l'arrel de l'arbre
      elem: Variable on s'ha de recollir l'element cercat
      ref: Dada que identifica l'element a cercar
   Retorna:
      0 : Recerca efectuada amb èxit
      1 : Element cercat no trobat
*/
   var res: natural; fivar /* per a emmagatzemar el resultat a retornar */
   si arbre == NUL llavors retorna 1; fisi
   opció
      cas és_menor (arbre^.inf, ref): res = recerca (arbre^.dre, elem, ref);
      cas és_major (arbre^.inf, ref): res = recerca (arbre^.esq, elem, ref);
      altrament: elem = arbre^.inf; res = 0;
   fiopció
   retorna res;
fifunció
```

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

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

Observeu el caràcter recursiu de l'algorisme.

2.2.2. Algorisme d'inserció

L'algorisme d'inserció és molt semblant al de recerca. Donat un element que volem inserir de manera ordenada en un arbre, i estant situats en un node, si l'element és més gran (segons el criteri d'ordenació) que el valor emmagatzemat en el node, caldrà inserir l'element en el subarbre dret (crida recursiva). Si, al contrari, l'element és més petit que el valor emmagatzemat en el node, caldrà inserir l'element en el subarbre esquerra (crida recursiva).

Observeu que es tracta d'un procés recursiu que finalitza quan es dona alguna de les dues situacions següents:

- l'element coincideixi amb el valor emmagatzemat en el node arrel de l'arbre actual, fet que indica que l'element ja existia a l'arbre,
- l'arbre actual sigui buit, fet que indica que estem en el subarbre on cal inserir el valor.

```
funció inserir (var arbre:^node, elem:T) retorna natural és
/* Arguments:
      arbre: Punter a l'arrel de l'arbre
      elem: Dada a inserir
   Retorna:
      0 : Inserció efectuada amb èxit
      1 : Error per manca de memòria
      2 : Ja existeix un element amb el mateix identificador
*/
   var res: natural; fivar /* per a emmagatzemar el resultat a retornar */
   si arbre == NUL
   llavors assignar_memòria (arbre);
           si arbre == NUL llavors retorna 1: fisi
           arbre^.esq = NUL; arbre^.dre = NUL; arbre^.inf = elem; retorna 0;
   sinó opció
          cas és_menor* (arbre^.inf, elem): res = inserir (arbre^.dre, elem);
          cas és_major* (arbre^.inf, elem): res = inserir (arbre^.esq, elem;
          altrament: res = 2;
        fiopció
        retorna res;
   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
```

2.2.3. Algorisme d'eliminació

L'algorisme d'esborrament és una mica més complicat que l'algorisme d'inserció. Evidentment, per a poder esborrar un element, primer cal efectuar la recerca, algorisme que ja hem comentat abans.

Una vegada trobat l'element, cal aplicar el mètode basat en els punts següents:

- Si l'element a esborrar es troba en una fulla, simplement s'elimina.
- Si l'element a esborrar es troba en un node que té un únic fill, el node que conté el valor a eliminar se substitueix pel node fill.
- Si l'element a esborrar es troba en un node que té dos nodes fills, el node que conté el valor a eliminar se substitueix adequadament pel node que està més a la dreta del subarbre esquerre o pel node que està més a l'esquerra del subarbre dret.

Comprovarem el funcionament dels passos anteriors. Suposem que volem eliminar el valor 10 de l'arbre de la figura 6.

Figura 6. Arbre binari de recerca on cal eliminar el valor 10

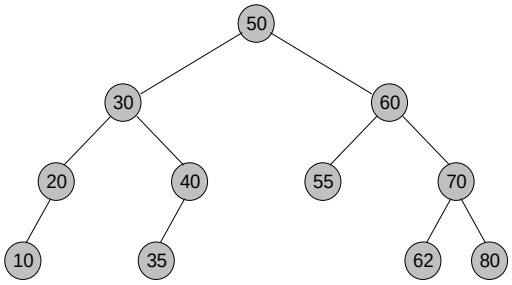
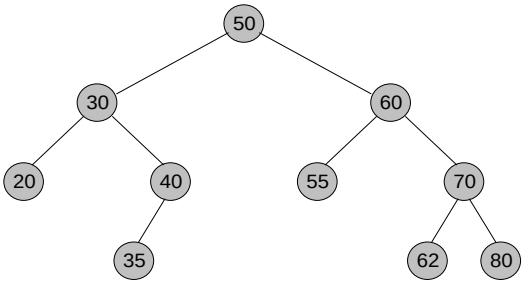


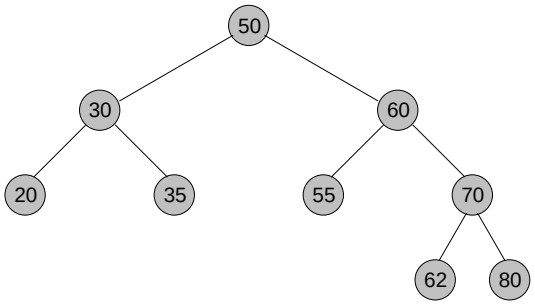
Figura 7. Arbre binari de recerca de la figura 6 una vegada eliminat el valor 10.



En efectuar la recerca el trobem en una fulla. En aquest cas, el node s'elimina i observem que l'arbre obtingut (figura 7) continua sent de recerca.

Suposem que volem eliminar el valor 40 en l'arbre de la figura 7. En efectuar la recerca el trobem en un node interior, el qual únicament té un fill. Segons les regles convingudes, se substitueix pel node fill i observem que l'arbre obtingut (figura 8) continua sent de recerca.

Figura 8. Arbre binari de recerca de la figura 7 una vegada eliminat el valor 40.



Suposem que volem eliminar el valor 60. Després de cercar-lo el trobem en un node interior que té dos fills. Segons les regles convingudes, cal substituir-lo pel node de més a l'esquerra del subarbre dret (node que conté el valor 62, obtenint l'arbre de la figura 9) o pel node de més a la dreta del subarbre esquerre (node que conté el valor 55, obtenint l'arbre de la figura 10).

Figura 9. Arbre binari de recerca de la figura 8 una vegada eliminat el valor 60 substituint-lo pel node de més a l'esquerra del subarbre dret.

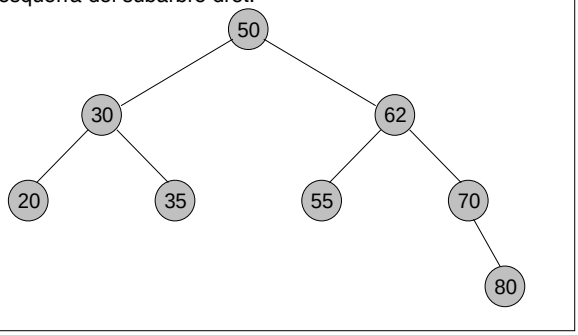
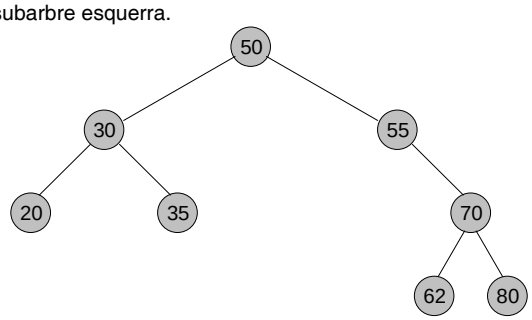


Figura 10. Arbre binari de recerca de la figura 8 una vegada eliminat el valor 60 substituint-lo pel node més a la dreta del subarbre esquerra.



Observem que tant en un cas (procés que obté l'arbre de la figura 9) com en l'altre (procés que obté l'arbre de la figura 10), els nodes escollits per a substituir el node que contenia el valor 60 (62 i 55 de la figura 8) no tenien cap fill. És clar que el node que contenia el valor 62 podria haver tingut fills per la dreta (i continuaria sent el node de més a l'esquerra del subarbre dret –figura 8–) a l'igual que el node que contenia el valor 55 podria haver tingut fills per l'esquerra (i continuaria sent el node de més a la dreta del subarbre esquerra –figura8–). Evidentment, els possibles subarbres fills dels nodes que contenen els valors 62 i 55 no poden desaparèixer i, per tant:

- En la figura 9, el possible subarbre dret del node que contenia el valor 62 a la figura 8, hauria d'haver passat a ser subarbre esquerra del node que conté el valor 70 (o mantenir-se com a subarbre dret del node que conté el valor 62 si el node eliminat no tenia subarbre dret).

- En la figura 10, el possible subarbre esquerra del node que contenia el valor 55 a la figura 8, hauria d'haver continuat sent el subarbre esquerra del node que conté el valor 55 (o passar a ser el subarbre dret del subarbre esquerra del node eliminat si hagués existit).

L'estructura de l'arbre resultant és diferent si s'executa l'una o l'altra opció, però en tots dos casos continua tractant-se d'un arbre binari de recerca. Quin resultat final us sembla més adequat? És més adequat el resultat final de la figura 9, ja que deixa un arbre més "ben repartit". Això no vol dir que s'hagi d'aplicar sempre l'elecció que ha possibilitat aquest resultat final, ja que aquest depèn de la situació de partida.

Així doncs, ja estem en condicions de presentar el pseudocodi de la funció esborrar en un arbre binari de recerca.

```
funció esborrar(var arbre:^node, var elem:T, ref:T*) retorna natural és
/* Arguments:
      arbre: Punter a l'arrel de l'arbre
      elem: Variable on s'ha de recollir l'element a esborrar
      ref: Dada de tipus T* que identifica l'element a esborrar
   Retorna:
      0 : Esborrat efectuat amb èxit
      1 : Error ja que no existeix un tal element identificat per ref
*/
   var res: natural; /* per a emmagatzemar el resultat a retornar */
       aux, auxbis: ^node;
   fivar
   opció
      cas arbre == NUL: res = 1;
      cas és_major(arbre^.inf, ref): res = esborrar (arbre^.esq, elem, ref);
      cas és_menor(arbre^.inf, ref): res = esborrar (arbre^.dre, elem, ref);
      altrament: /* l'element a esborrar està en el node apuntat per arbre */
         aux = arbre; res = 0; element = arbre^.inf;
         opció
            cas aux^.esq == NUL i aux^.dre == NUL: /* és una fulla */
               arbre = NUL; alliberar_memòria (aux);
            cas aux^.esq == NUL i aux^.dre != NUL: /* només fill dret */
               arbre = aux^.dre; alliberar_memòria (aux);
            cas aux^.esq != NUL i aux^.dre == NUL: /* només fill esquerre */
               arbre = aux^.esq; alliberar_memòria (aux);
            altrament: /* dos fills */
            /* substituïm pel node més a la dreta del subarbre de l'esquerra */
               auxbis = aux^.esq;
               mentre auxbis^.dre != NUL fer
                  aux = auxbis; auxbis = auxbis^.dre;
               fimentre
               si aux == arbre llavors aux^.esq = auxbis^.esq;
               sinó aux^.dre = auxbis^.esq
               fisi
               arbre^.inf = auxbis^.inf; alliberar_memòria (auxbis);
         fiopció
   fiopció
   retorna res;
fifunció
```

Observeu, altra vegada, el caràcter recursiu de l'algorisme. És important remarcar, també, que en cas de substitució per l'element que es troba més a la dreta del subarbre esquerre no se substitueix node per node, sinó element a esborrar per element del node més a la dreta del subarbre esquerre.

2.3. Arbres balancejats

Els arbres binaris de recerca permeten realitzar de manera eficient les operacions de recerca, eliminació i inserció, però tenen un defecte: el creixement i/o decreixement descontrolats, que provoquen que no quedin "ben repartits".

Observeu els exemples de les figures 11 i 12.

Figura 11. Arbre binari de recerca després de la seqüència d'insercions dels valors 22, 33, 44 i 55.

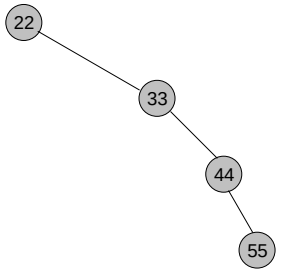
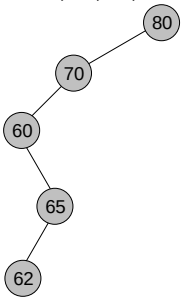


Figura 12. Arbre binari de recerca després de la seqüència d'insercions dels valors 80, 70, 60, 65 i 62



Oi que no estan "ben repartits"? Això provoca que la recerca de valors no aprofiti l'eficiència que el seu disseny aporta en l'ús del procés dicotòmic.

Per a solucionar aquesta situació apareix el concepte d'arbre balancejat.

> Un arbre balancejat és un arbre binari de recerca tal que, per a qualsevol node, les altures dels subarbres dret i esquerre no poden diferir en més d'una unitat.

La idea central és la d'arreglar l'arbre després de les insercions i les eliminacions de manera que es mantingui l'equilibri. Per a poder mantenir aquest equilibri cal tenir informació referent a l'equilibri de cada node de l'arbre. Això provoca l'aparició del concepte factor d'equilibri d'un node.

El factor d'equilibri d'un node és el resultat de la resta de l'altura del subarbre dret menys l'altura del subarbre esquerre.

Perquè l'arbre sigui balancejat cal mantenir en tots els nodes el factor d'equilibri en els valors 1, 0 o –1. Aquest factor s'emmagatzema en cada node i es recalcula, de manera adequada, en cada inserció i eliminació. En cas de pèrdua d'equilibri en algun node es reestructura l'arbre per a mantenir-lo com a arbre balancejat.

De l'apartat anterior es dedueix que la implementació d'un node per a arbre balancejat necessita un nou apartat:

```
tipus node = tupla
            inf : T; /* conté la informació */
            esq, dre : ^node; /* punters als dos fills */
            fe : enter;
          fitupla
```

El nou camp `fe` hauria de valer sempre `-1`, `0` o `1`, però en determinats moments pot valer `-2` o `2`, a causa de la pèrdua momentània de l'equilibri.

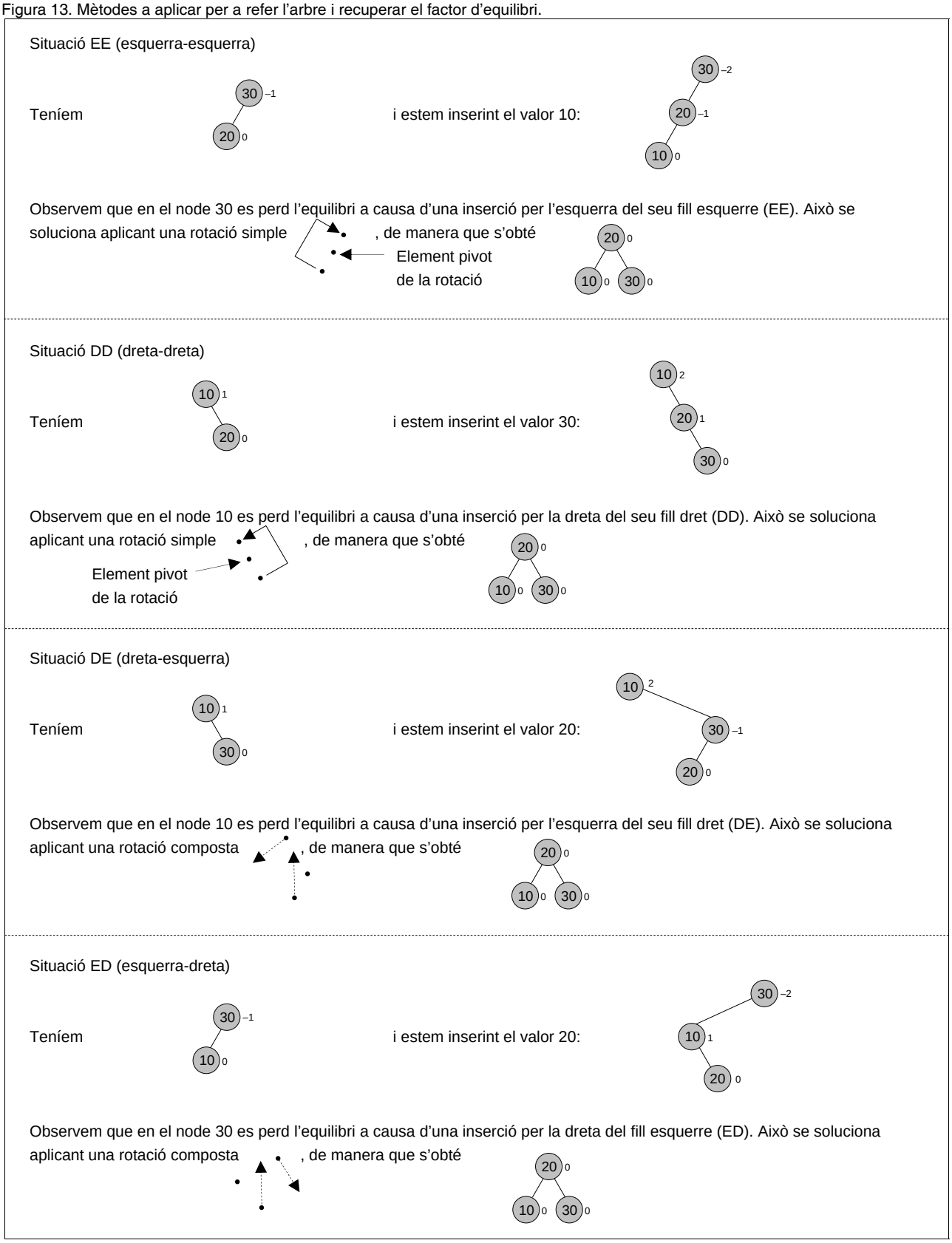
Presentem ara el funcionament dels algorismes d'inserció i esborrat per als arbres balancejats. No arribarem a escriure'n el pseudocodi. Ho deixem per a vosaltres. Us hi animeu? L'algorisme de recerca no canvia respecte als arbres binaris de recerca.

2.3.1. Algorisme d'inserció

En primer lloc, s'insereix l'element, segons l'algorisme per a arbres binaris de recerca (sempre s'insereix en una fulla), calculant-ne el factor d'equilibri (FE, a partir d'ara) que, evidentment, serà zero. Cal tornar enrere pel camí de recerca d'aquest nou node i recalcular els FE dels nodes pels quals es passa, fins a trobar un node que perdi l'equilibri o fins a arribar a l'arrel sense que perdi l'equilibri. Si es localitza un node amb pèrdua d'equilibri, cal refer l'arbre en aquest node i no cal continuar recalculant el FE per a la resta de nodes, ja que aquest mètode ens assegura que no haurà variat.

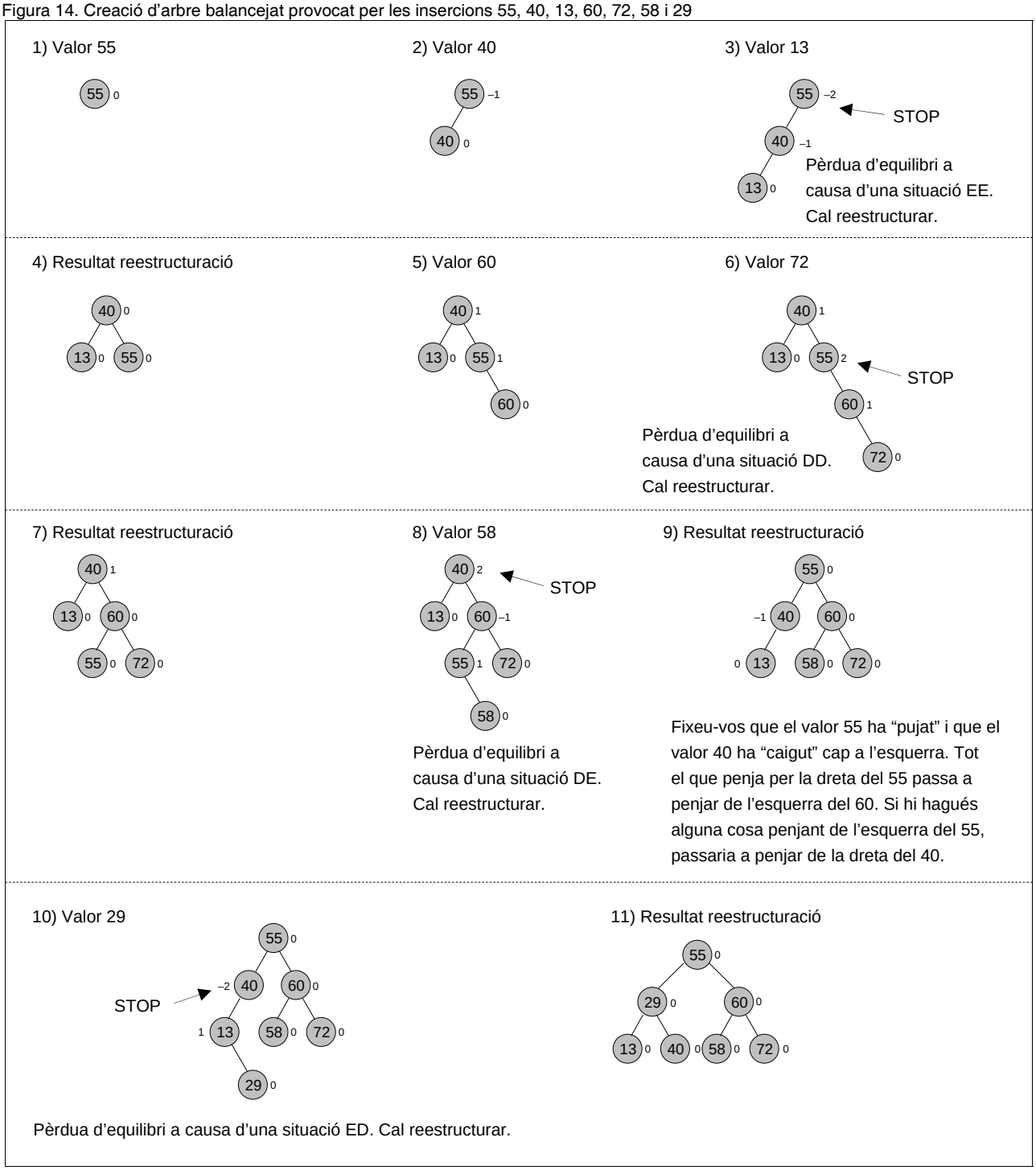
Per a refer l'arbre cal aplicar el mètode adequat segons sigui la situació produïda, que pot ser una de les quatre que mostra la figura 13.

Figura 13. Mètodes a aplicar per a refer l'arbre i recuperar el factor d'equilibri.



No hi ha res millor que un exemple per a entendre les diferents situacions que es poden presentar. Volem construir un arbre balancejat a partir de la seqüència de valors 55 - 40 - 13 - 60 - 72 - 58 - 29. La figura 14 ens mostra els diferents passos que hem de seguir.

Figura 14. Creació d'arbre balancejat provocat per les insercions 55, 40, 13, 60, 72, 58 i 29



2.3.2. Algorisme d'eliminació

L'algorisme per eliminar consisteix en l'algorisme d'esborrat en arbres binaris de recerca mantenint l'equilibri de l'arbre: una vegada eliminat un node, es retorna pel camí de recerca d'aquest nou node fins a l'arrel, recalculant els FE dels nodes pels quals es passa fins a arribar a l'arrel. Cal reestructurar l'arbre en aquest node en el qual s'ha perdut l'equilibri. A diferència de l'algorisme d'inserció, cal comprovar l'FE fins

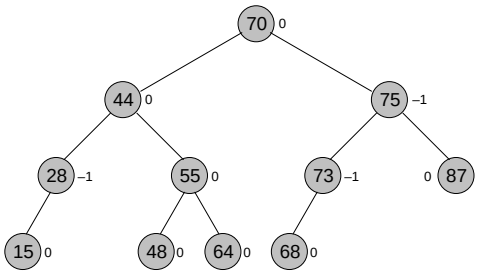
a l'arrel, ja que hi pot haver diversos nodes en els quals es perdi l'equilibri.

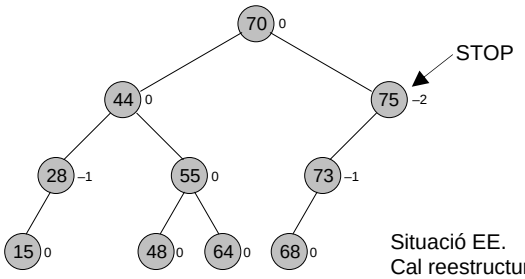
La figura 15 ens mostra diverses situacions que es presenten en eliminar valors en un arbre balancejat.

Figura 15. Processos d'eliminació de valors en arbres balancejats

1) Eliminar el valor 87

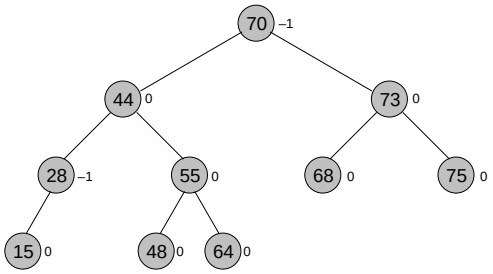
En ser una fulla, s'elimina i es recalcula el FE.





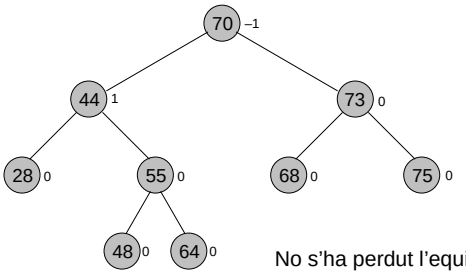
Situació EE.
Cal reestructurar.

2) Resultat reestructuració



3) Eliminar el valor 15

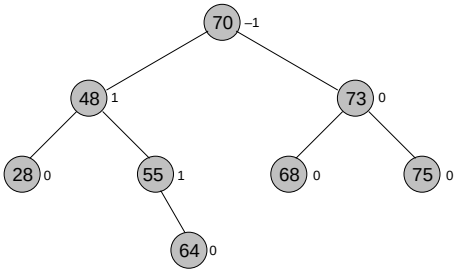
En ser una fulla, s'elimina i es recalcula el FE.



No s'ha perdut l'equilibri.

4) Eliminar el valor 44

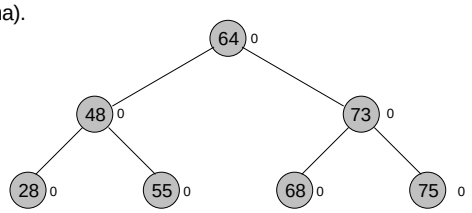
És un node interior amb dos fills. Tenim dues opcions. Decidim substituir-lo per l'element més a l'esquerra del fill dret, ja que tenia FE =1 (més altura arbre dret).



No s'ha perdut l'equilibri

5) Eliminar el valor 70

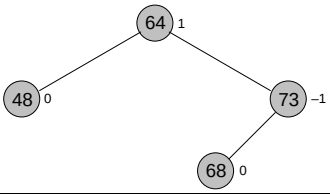
És un node interior (l'arrel) amb dos fills. Com que FE = -1, decidim substituir-lo pel valor que hi ha més a la dreta del subarbre esquerre (podríem haver escollit una altra forma).



No s'ha perdut l'equilibri

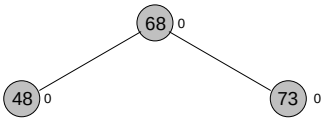
6) Eliminar els valors 75-28-55

Són fulles que s'eliminen (una per una) i en cap cas es perd l'equilibri.



7) Eliminar el valor 64

És un node interior (l'arrel) amb dos fills. Com que FE = 1, decidim substituir-lo pel valor més a l'esquerra del subarbre dret.



L'anàlisi matemàtica dels algorismes d'inserció i eliminació demostra que tots són d'ordre logarítmic. D'altra banda, diverses anàlisis mostren que són més freqüents les rotacions en les operacions d'inserció que en les d'eliminació.

2.4. Arbres multicamí

Les variants d'arbres binaris vistes fins ara són desenvolupades per funcionar en la memòria principal. Hi ha aplicacions, però, que gestionen grans volums d'informació que no caben en la memòria principal i que necessiten emmagatzemar-se en arxius perquè puguin ser localitzats de forma eficient. D'aquesta manera, apareixen les tècniques d'indexació d'arxius, les quals es basen en arbres (un per a cada índex) que emmagatzemen la clau i l'adreça on es troben les dades (arxius de dades). És a dir, la part d'informació d'un node es compon de la clau i l'adreça. A part, hi ha els punters (dos en tractar-se d'arbres binaris) i l'FE (si són arbres balancejats). Cada node s'emmagatzema en un registre de l'arxiu índex.

Suposem que indexéssim un arxiu d'1.000.000 de registres amb un arbre binari balancejat. Necessitarem, per a cercar un element, uns 20 accessos a disc (logaritme en base 2 d'1.000.000), ja que cada accés a un nou node implica accés a un nou registre de l'arxiu índex. Ara bé, si l'arbre estigués organitzat en pàgines, cadascuna de les quals contingués, com a mínim, 100 elements, l'ordre d'accessos baixaria espectacularment a logaritme en base 100 d'1.000.000, és a dir, 3 accessos a disc. Per aquest motiu entren en acció els arbres multicamins.

> Un arbre multicamí és un arbre balancejat que conté, en un node, diversos valors i diversos punters a altres nodes. Si considerem que conté N valors, haurà de contenir N + 1 punters.

Gràficament, un node amb 4 elements tindria l'aparença següent:

P1	Inf 1	P2	Inf 2	P3	Inf 3	P4	Inf 4	P5

Les caselles Inf1, Inf2, Inf3 i Inf4 contindrien els valors i les caselles P1, P2, P3, P4 i P5 contindrien els punters a nous nodes. Des d'un node es pot tenir accés a cinc nodes diferents, és a dir, disposem de cinc camins a seguir. ¿Es veu clar ara d'on prové el nom d'arbres multicamins?

Hi ha diferents tipus d'arbres multicamins. En presentarem dos de molt coneguts: els arbres B i els arbres B^+.

2.4.1. Arbres B

Els arbres B són una generalització dels arbres balancejats per a aconseguir arbres multicamins. Vegem-ne la definició.

Autors dels arbres B

El disseny dels arbres B és obra de Bayer i McCreight el 1970. El seu nom prové de Bayer? És una incògnita.

Un arbre B d'ordre d és un arbre que té les característiques següents:

- Cada node, excepte l'arrel, conté entre d i 2d elements.
- Cada node, excepte l'arrel i les fulles, té entre d + 1 i 2d + 1 fills.
- El node arrel o no té fills (perquè és una fulla) o en té, com a mínim, dos.
- Totes les fulles es troben en un mateix nivell (balancejat perfecte).
- Els elements emmagatzemats en un node estan ordenats de més petit a més gran (d'esquerra a dreta).
- Cada element d'un node que no és fulla té dos fills. Tots els elements del seu subarbre esquerre són més petits que ell i tots els elements del seu subarbre dret són més grans que ell.

Per a poder treballar amb arbres B, cal que a cada node hi hagi informació referent al nombre de dades que està emmagatzemant en un moment donat. Hi ha diferents maneres de poder tenir aquesta informació:

- Mitjançant una dada afegida que permet obtenir aquesta informació, de manera que l'estructura d'un node seria semblant a aquesta:

P_1	Inf_1	P_2	Inf_2	P_3		P_{2d}	Inf_{2d}	P_{2d+1}	X

  Es pot observar que tenim 2d + 1 punters (P_1, P_2, P_3,..., P_{2d+1}) i 2d valors (Inf_1, Inf_2,..., Inf_{2d}). La marca X correspondria a la quantitat de valors que en un determinat moment conté el node, que exceptuant l'arrel ha de ser sempre entre d i 2d.

- Sense una dada afegida i calculant el valor corresponent com a quantitat, disminuïda en una unitat, de punters amb valor no nul.

Aquest funcionament és possible en els nodes interiors, però no en les fulles, ja que en aquestes tot punter és nul. De fet, podríem considerar estructures de nodes diferents per als nodes interiors i per a les fulles. En les fulles no fan falta els punters, però en canvi sí que és necessari saber quants valors estan emmagatzemant en un moment donat.

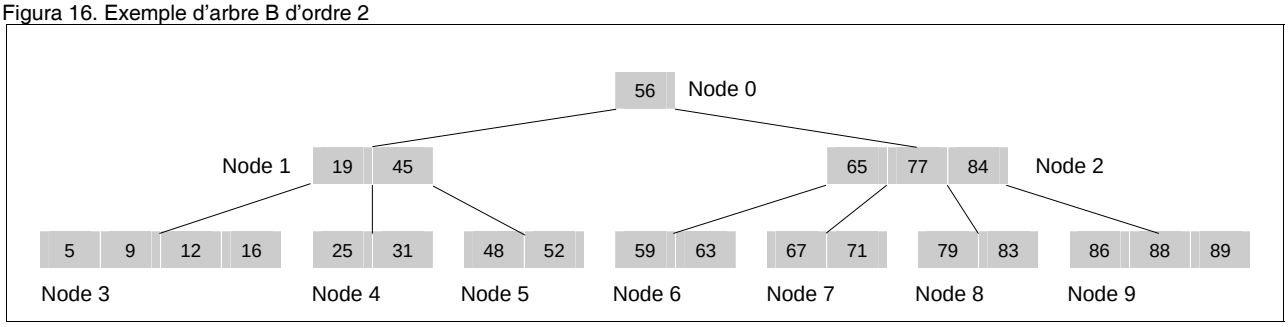
Algorisme de recerca

L'algorisme de recerca per a arbres multimcamins no és altra cosa que una generalització de l'algorisme de recerca per a arbres binaris. Vegem-ne el pseudocodi.

```
funció recerca (arbre:^node, var elem:T, ref: T*) retorna natural és
/* Arguments:
      arbre: Punter a l'arrel de l'arbre
      elem: Variable on s'ha de recollir l'element cercat
      ref: Dada que identifica l'element a cercar
   Retorna:
      0 : Recerca efectuada amb èxit
      1 : Element cercat no trobat
*/
   var res: natural; fivar /* per a emmagatzemar el resultat a retornar */
   si arbre == NUL llavors retorna 1; fisi
   opció
      cas és_inclòs_en_el_node (arbre, ref, elem): res = 0;
      cas és_major (arbre^.inf1, ref): res = recerca (arbre^.P1, elem, ref);
      cas és_menor (arbre^.inf1, ref): res = recerca (arbre^.P2, elem, ref);
      cas és_menor (arbre^.inf2, ref): res = recerca (arbre^.P3, elem, ref);
      ...
      cas és_menor (arbre^.inf2d, ref): res = recerca (arbre^.P2d+1, elem, ref);
   fiopció
   retorna res;
fifunció

funció és_inclòs_en_el_node (p: ^node, ref: T*, var elem: T) retorna booleà és
   /* Codi que comprovi si en el node apuntat per p hi ha un element amb part T* igual a
ref. En aquest cas retorna cert i copia l'element en el paràmetre elem. En cas contrari,
retorna fals. */
fifunció
```

Figura 16. Exemple d'arbre B d'ordre 2



La figura 16 ens mostra la representació gràfica d'un arbre B d'ordre 2. Primer de tot, observem que a cap node hi representem les caselles corresponents a punters. Cada fletxa parteix d'una casella punter i apunta el node corresponent. D'altra banda, hem de tenir clar que cada node conté fins a 4 caselles informació, però només representem les caselles que contenen valor. Per tant, veiem nodes amb 1, 2, 3 o 4 valors.

Observem, també, que per a poder fer el seguiment corresponent de l'algorisme de recerca hem numerat els nodes com a $Node_0$, $Node_1$,..., $Node_9$. En la realitat, els nodes no estan numerats.

1) Recerca de l'element 77

- Visitem l'arrel (node 0) i, en comparar el valor cercat 77 amb el valor 56 del node, com que 77 és més gran que 56, seguim pel punter (fletxa) de la dreta del valor 56, la qual ens porta al node 2.
- Visitem el node 2, on trobem el valor cercat.

2) Recerca de l'element 79

- Visitem l'arrel (node 0), el qual ens envia al node 2.
- Visitem el node 2, on posicionem el valor cercat 79 entre els valors 77 i 84, la qual cosa ens porta a seguir el camí cap al node 8.
- Visitem el node 8, on trobem el valor cercat.

3) Recerca de l'element 80

- Visitem l'arrel (node 0), el qual ens envia al node 2.
- Visitem el node 2, el qual ens envia al node 8.
- Visitem el node 8, on no hi ha l'element cercat. Ja som en una fulla. La recerca ha finalitzat sense trobar l'element desitjat.

Algorisme d'inserció

Els arbres B creixen de baix cap a dalt: de les fulles cap a l'arrel. L'algorisme consisteix, en primer lloc, a aplicar l'algorisme de recerca per a situar-se en el node fulla, on hauria d'haver-hi l'element a inserir. Una vegada en el node, hi ha diferents possibilitats:

1) Si hi ha espai (el node conté menys de $2d$ elements), s'insereix adequadament (mantenint l'ordre dins la fulla) i l'algorisme finalitza.

2) Si no hi ha espai (el node conté exactament $2d$ elements), el node fulla es divideix en dos (en realitat, es crea dinàmicament un nou node), i els $2d + 1$ elements (els $2d$ que ja existien a la fulla, més l'element que

cal inserir) es distribueixen ordenadament entre els dos nodes fulla, amb d valors a cada node fulla (els d valors més petits en el node fulla esquerre i els d valors més grans en el node fulla dret). Aleshores l'element central puja cap al node pare del node fulla inicial i s'hi insereix adequadament. La inserció en el node pare consisteix a repetir el mateix procés, cosa que pot provocar la creació d'un nou node amb la pujada del valor central cap al pare corresponent. Quan el node pare en què cal inserir un element és el node arrel de l'arbre i ja no hi ha espai disponible, aquest provoca la creació de dos nodes nous (en lloc de només un!), ja que l'un és el corresponent al procés normal i l'altre, al nou node arrel (el node pare al qual puja l'element central).

Terminologia anglesa

El procés descrit consistent en la divisió d'un node en dos, repartint-se els valors i passant el valor central al node pare, s'anomena, en terminologia informàtica anglesa, *split*. Per això és molt normal indicar, quan cal aplicar el procés sobre un node A, que el node A e*splita*, en un clar abús de terminologia. Però estareu d'acord que dir "el node A *esplita*" és molt clar i molt més simple que dir "es crea un nou node que recull la meitat superior del conjunt ordenat de valors format pels valors del node A més el valor a inserir; el node A es queda amb la meitat inferior del mateix conjunt de valors i el valor central s'insereix en el node pare del node A". Per tant, utilitzarem l'expressió "el node *esplita*" per a fer referència a tot aquest procés.

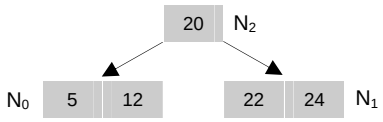
Vegem-ne un exemple per a entendre les diferents situacions que es poden presentar. Volem construir un arbre B d'ordre 2 a partir de la seqüència de valors 5 - 22 - 24 - 12 - 20 - 16 - 10 - 26 - 8 - 46 - 15 - 19 - 43 - 14 - 55 - 30 - 61. La figura 17 ens mostra els passos a seguir.

Figura 17. Creació d'un arbre B amb els valors 5, 22, 24, 12, 20, 16, 10, 26, 8, 46, 15, 19, 43, 14, 55, 30 i 61

D'entrada, cada un dels quatre primers valors permet anar omplint el node arrel de manera adequada. S'obté:

L'arbre és format per un únic node i és totalment ple.

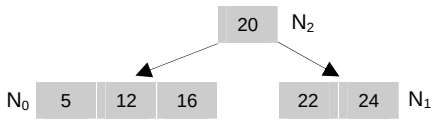
Inserció del valor 20
No hi cap. Cal *esplitar*. Com que és l'arrel, implica la creació de dos nodes:



Dóna la casualitat que l'element inserit és el que "puja" cap al pare. Recordeu que sempre "puja" l'element central, que pot coincidir o no amb l'element inserit. Numerarem els nodes per a fer un millor seguiment.

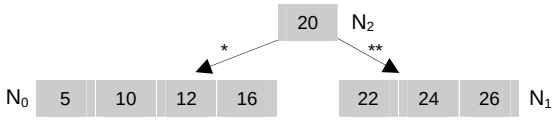
Inserció del valor 16

Sense cap problema, el posem en el node zero.



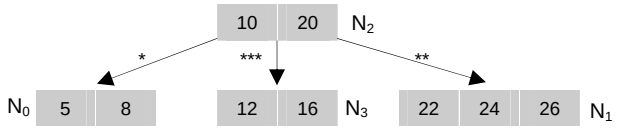
Inserció dels valors 10 i 26

De la mateixa manera, no tenim cap problema per a situar-los en els nodes 0 i 1 respectivament:



Inserció del valor 8

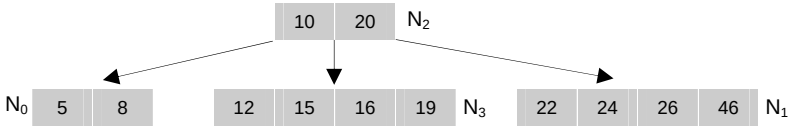
Correspon situar-lo en el node 0, però no hi cap. Cal *esplitar*.



Observem que en aquest procés s'han creat el node N_3 i el moviment del punter ** en el node pare. Així mateix, també s'ha creat el punter ***.

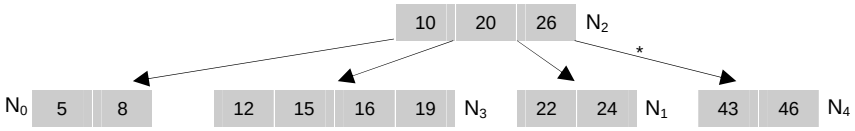
Inserció dels valors 46, 15, 19

Cap problema. Les diferents insercions s'efectuen en els nodes 1,3 i 3, de manera que obtenim:



Inserció del valor 43

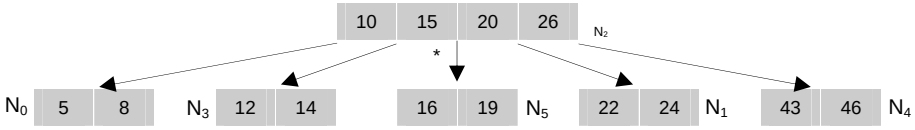
Correspon situar-lo en el node 1, però no hi cap. Cal *esplitar*.



S'han creat el node N_4 i el punter *. En aquest cas no s'ha hagut de moure cap punter dels existents a N_2.

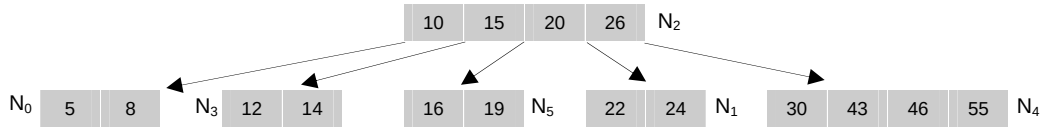
Inserció del valor 14

Correspon situar-lo en el node 3, però no hi cap. Cal *esplitar*.

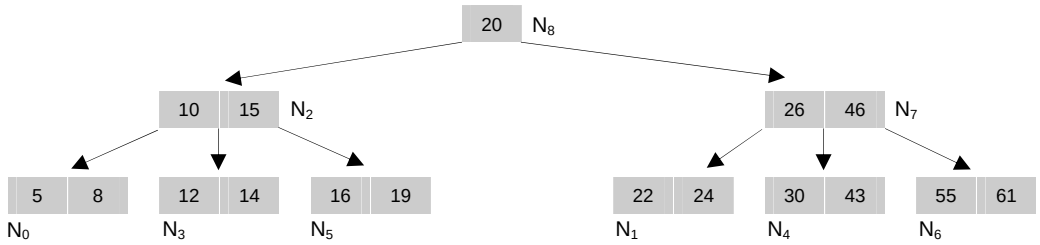


S'han creat el node N_5 i el punter *. Els valors 20 i 26, juntament amb els punters de la seva dreta, s'han hagut de desplaçar una posició a la dreta.

Inserció dels valors 55 i 30
Cap problema. S'insereixen en el node N_4.



Inserció del valor 61
Correspon al node N_4, però no hi cap. Cal *esplitar*. Aquest *split* implicarà que el valor 46 pugi al pare (node arrel), però en aquest tampoc no hi ha espai, la qual cosa implica un nou *split*. Tenim, per tant:



Observeu que s'han creat els nodes N_6, N_7 i N_8 i que hi ha hagut, també, un fort moviment de punters.

Algorisme d'eliminació

L'algorisme per eliminar ha de fer quelcom contrari a l'algorisme d'inserció, ja que un node (excepte l'arrel) mai no pot quedar amb menys de d valors. Per tant, hi haurà algun procés invers a l'split, consistent en una fusió de nodes. En efecte, l'algorisme consisteix a cercar l'element a esborrar i, una vegada localitzat, hi ha dues possibilitats:

1) Si el valor a esborrar es troba en una fulla, se suprimeix.

2) Si el valor a esborrar es troba en un node no fulla, se substitueix pel valor que es troba més a la dreta del subarbre apuntat pel seu punter esquerre o pel valor que es troba més a l'esquerra del subarbre apuntat pel seu punter dret. Aquest valor sempre es troba en una fulla.

En qualsevol dels dos casos, s'ha de verificar si la fulla que ha acabat perdent el valor continua tenint d valors com a mínim. Si és així, no s'ha de fer res més. En cas contrari, s'ha d'aplicar una reorganització o fusió de la fulla amb la fulla germana de la dreta, si existeix, o amb la fulla germana de l'esquerra (si no tenia germana per la dreta). Tota fulla que no sigui arrel sempre té una fulla germana.

– Efectuarem una reorganització quan la fulla germana tingui més de d valors. En aquest cas, aquesta cedirà valors a la fulla que ha perdut

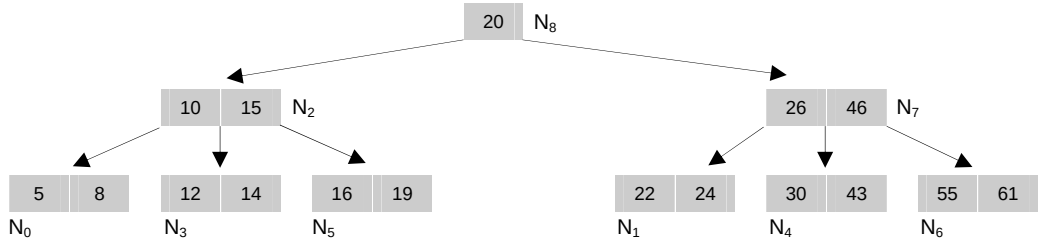
l'element suprimit. Aquesta cessió comporta un moviment del valor mitjà existent en el pare de les dues fulles.

– Efectuarem una fusió quan la fulla germana tingui exactament d valors. En aquest cas, passa a haver-hi una única fulla que conté els valors de les dues més el valor mitjà que residia en el pare. Per tant, el pare es queda amb un valor menys i cal aplicar-hi el mateix procés de comprovació que sobre la quantitat de valors que conté. Si és inferior a d cal aplicar-hi el mateix procés de reorganització o fusió.

Vegem exemples d'eliminació sobre el darrer arbre de la figura 17. Suposem que volem eliminar els valors 20 - 19 - 24 - 22 - 43 - 14 - 46 - 16. La figura 18 ens en mostra els passos a seguir.

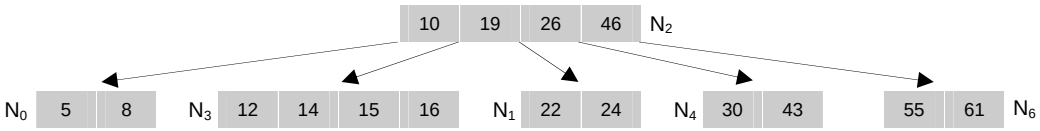
Figura 18. Exemples d'eliminació de valors en un arbre B

Partim del següent arbre B d'ordre 2



Eliminar el valor 20

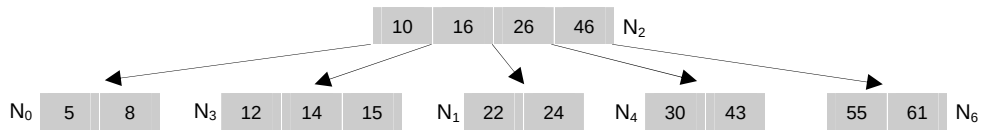
No és en una fulla. El substituirem per l'element que està més a la dreta del subarbre esquerre. És el valor 19 que es troba en el node N_5. La fulla N_5 queda amb menys de dos elements. No té germana per la dreta, però sí que té germans per l'esquerra (node N_3). Com que aquest té exactament dos valors, ens veiem obligats a fusionar N_3 amb N_5 baixant el valor 15 del pare N_2. Aquest node queda, a la vegada, amb menys de dos valors. Cal aplicar-hi el procés de reorganització o fusió. Té el node N_7 germà per la dreta amb dos valors. Cal aplicar una fusió: cal unir N_2 amb N_7 i baixar el valor 19 del pare N_8 (inicialment hi havia el valor 20, però l'hem substituït pel valor 19). En baixar l'element del node N_8 aquest desapareix, és a dir, l'arbre disminueix en una unitat la seva altura.



Observem que s'han eliminat els nodes N_5, N_7 i N_8.

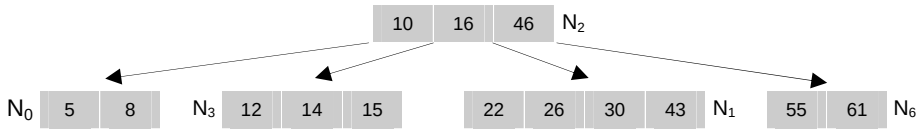
Eliminar el valor 19

És en un node interior i el substituïm pel valor que està més a la dreta del subarbre esquerre, és a dir, l'anterior immediatament a ell segons el criteri d'ordenació. És l'element 16. Obtenim que:



Eliminar el valor 24

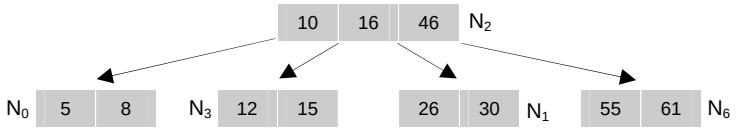
Es troba en un node fulla (N1). L'eliminem i en quedar-se amb menys de dos elements ens cal fusionar amb la germana dreta (N4), que té exactament dos valors. Obtenim que:



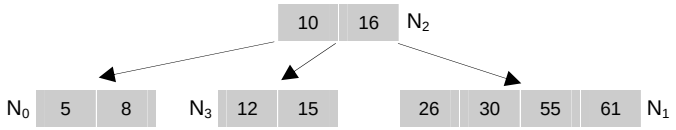
Observem que s'ha eliminat el node N4

Eliminar els valors 22, 43 i 14

Com que són valors que estan en fulles, els eliminem, i en cap cas cal reorganitzar ni fusionar ja que totes queden amb dos valors com a mínim.



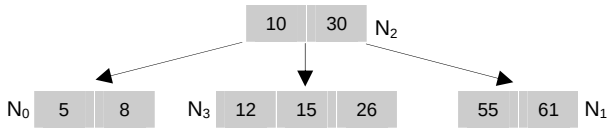
Eliminar el valor 46



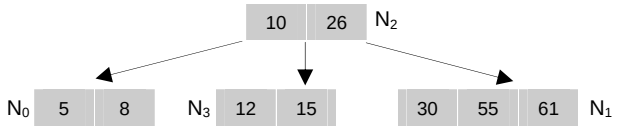
Observem que el node N_6 ha desaparegut.

Eliminar el valor 16

El substituïm pel valor immediatament anterior:15. El node N_3 queda amb menys de dos valors i té una germana per la dreta amb més de dos valors. Reorganitzem-lo.



El resultat de la reorganització també hauria pogut ser:



2.4.2. Arbres B⁺

Els arbres B^+ són molt semblants als arbres B. La característica principal d'aquests arbres és que tots els valors es troben a les fulles. Per tant:

– Qualsevol camí des de l'arrel per a arribar a un element té la mateixa longitud.

– En els nodes interiors no cal tenir emmagatzemats els elements sencers, que es trobaran en les fulles. Només cal tenir-hi emmagatzemats els valors de classificació.

Un arbre B^+ d'ordre `d` és un arbre que verifica que:

- Cada node, excepte l'arrel, conté entre `d` i `2d` valors de classificació.
- Cada node, excepte l'arrel i les fulles, té entre `d + 1` i `2d + 1` fills.
- El node arrel o no té fills (perquè és una fulla) o en té, com a mínim, dos.
- Totes les fulles es troben a un mateix nivell (balancejat perfecte) i són les que contenen els elements que emmagatzema l'arbre.
- Els valors emmagatzemats dins un node estan ordenats de més petit a més gran (d'esquerra a dreta).
- Cada valor d'un node interior té dos fills. Tots els valors del seu subarbre esquerre són més petits que ell i tots els valors del seu subarbre dret són més grans o iguals que ell.

Observeu que a diferència dels arbres B, en els quals tots els elements del subarbre dret de qualsevol element són més grans que l'element en qüestió, en els arbres B^+ tots els valors del subarbre dret de qualsevol valor són més grans o iguals que el valor en qüestió.

Observeu, també, que en el paràgraf anterior utilitzem el terme element per fer referència al contingut dels nodes dels arbres B i el terme valor per fer referència al contingut dels nodes dels arbres B^+, ja que en aquests els elements només són en els nodes fulles.

En els arbres B^+ s'evita la costosa operació de reorganització, però això ja ho comprovarem en l'algorisme d'eliminació corresponent.

Algorisme de recerca

És molt similar al dels arbres B, però amb la particularitat que en trobar el valor cercat en un node interior, cal continuar la recerca pel node apuntat per la dreta.

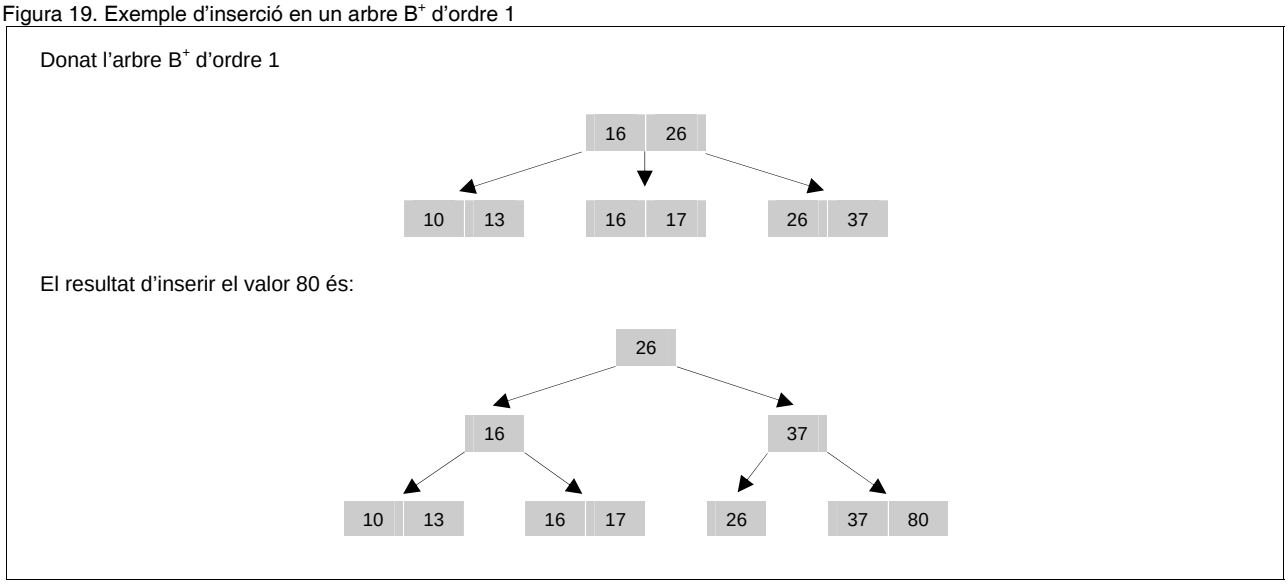
El fet que un valor de recerca existeixi en un node interior no implica que hi hagi un element amb aquell valor en una de les fulles; simplement vol dir que en algun moment de la vida de l'arbre sí que hi havia existit. Podrem constatar aquest fet en aplicar l'algorisme d'eliminació.

Algorisme d'inserció

És molt similar al dels arbres B, però amb la particularitat que quan una fulla no té més espai i s'aplica el procés de split, l'element central es manté a la nova fulla (que quedarà amb d+1 valors) i una còpia del seu valor de classificació passa a afegir-se al node pare de la fulla inicial. En els arbres B, era l'element central –i no pas el valor de classificació– el que passava a formar part del node pare. En canvi, l'aplicació del procés de split a un node interior és idèntica a la que s'aplica en els arbres B.

La figura 19 ens en mostra un exemple.

Figura 19. Exemple d'inserció en un arbre B^+ d'ordre 1



Algorisme d'eliminació

És molt més simple que en arbres B. Això és degut al fet que els elements a eliminar sempre es troben a les fulles i no hi ha cap problema que el valor de classificació corresponent a l'element eliminat

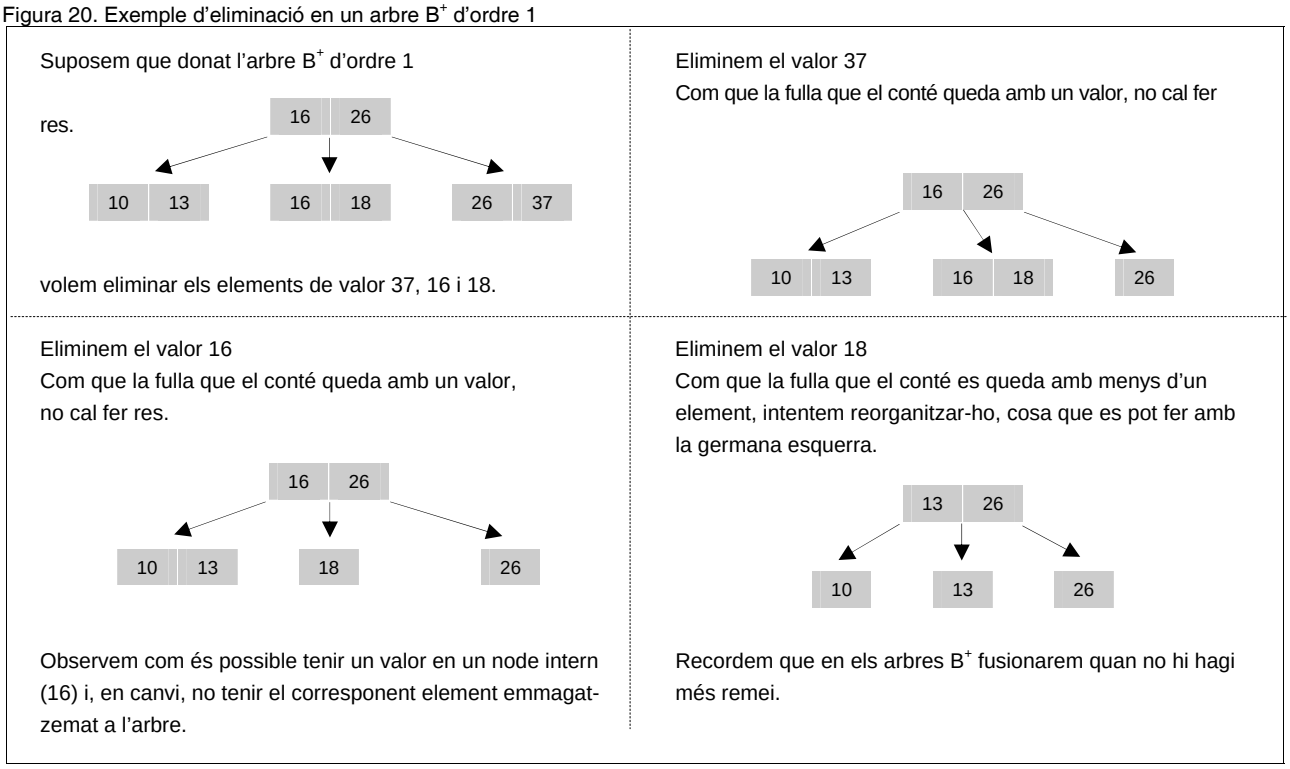
continuï estant en nodes interiors, tal com havíem comentat en l'algorisme de recerca. Tenim dues possibilitats:

1) Si en eliminar un element la fulla queda amb d valors com a mínim, no es fa res més, encara que en algun node interior o en el node arrel hi hagi una còpia del valor de classificació de l'element eliminat.

2) Si en eliminar un element la fulla queda amb menys de d valors, cal intentar fer una reorganització de valors en les fulles i en l'índex. Caldrà fer fusió quan no hi hagi més remei.

La figura 20 ens en mostra un exemple.

Figura 20. Exemple d'eliminació en un arbre B^+ d'ordre 1



2.4.3. Utilitat en arxius seqüencials indexats

En aquests moments coneixem diferents tipus d'arbres, des dels més genèrics fins als més específics (arbres B i B^+). Quan s'utilitzen? Ja sabem que els arbres tenen diverses aplicacions, una de les quals és la gestió dels índexs en els arxius seqüencials indexats.

La figura 21 ens mostra un arxiu de dades amb diferents camps pel que es desitja disposar d'accés directe i seqüencial per valor de dos camps. Per aconseguir-ho tenim la necessitat de dos índexs, tal i com s'observa a la figura 21.

Figura 21. Exemple d'un arxiu de dades indexat per dos camps diferents

Arxiu de dades

	Número	Cognom	Nom	Data naixement	Pes	Alçada	Sexe
1	7893	Ollé	Teresa	04.12.1985	45	150	D
2	4536	Esteve	Guifré	12.04.1980	60	170	H
3	2375	Miralles	Oriol	24.05.1979	62	173	H
4	9823	Brito	Júlia	23.09.1983	50	165	D
5	1287	Pelfort	Anna	15.07.1987	53	160	D
6	7834	Coromina	Jaume	10.10.1980	70	175	H

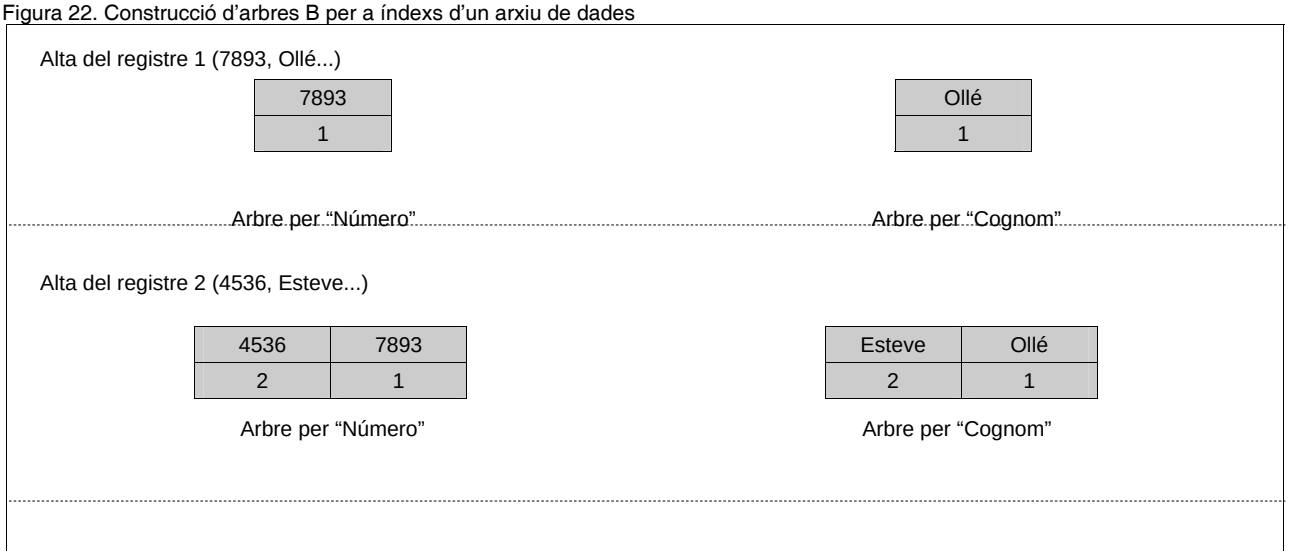
Arxius d'índexs

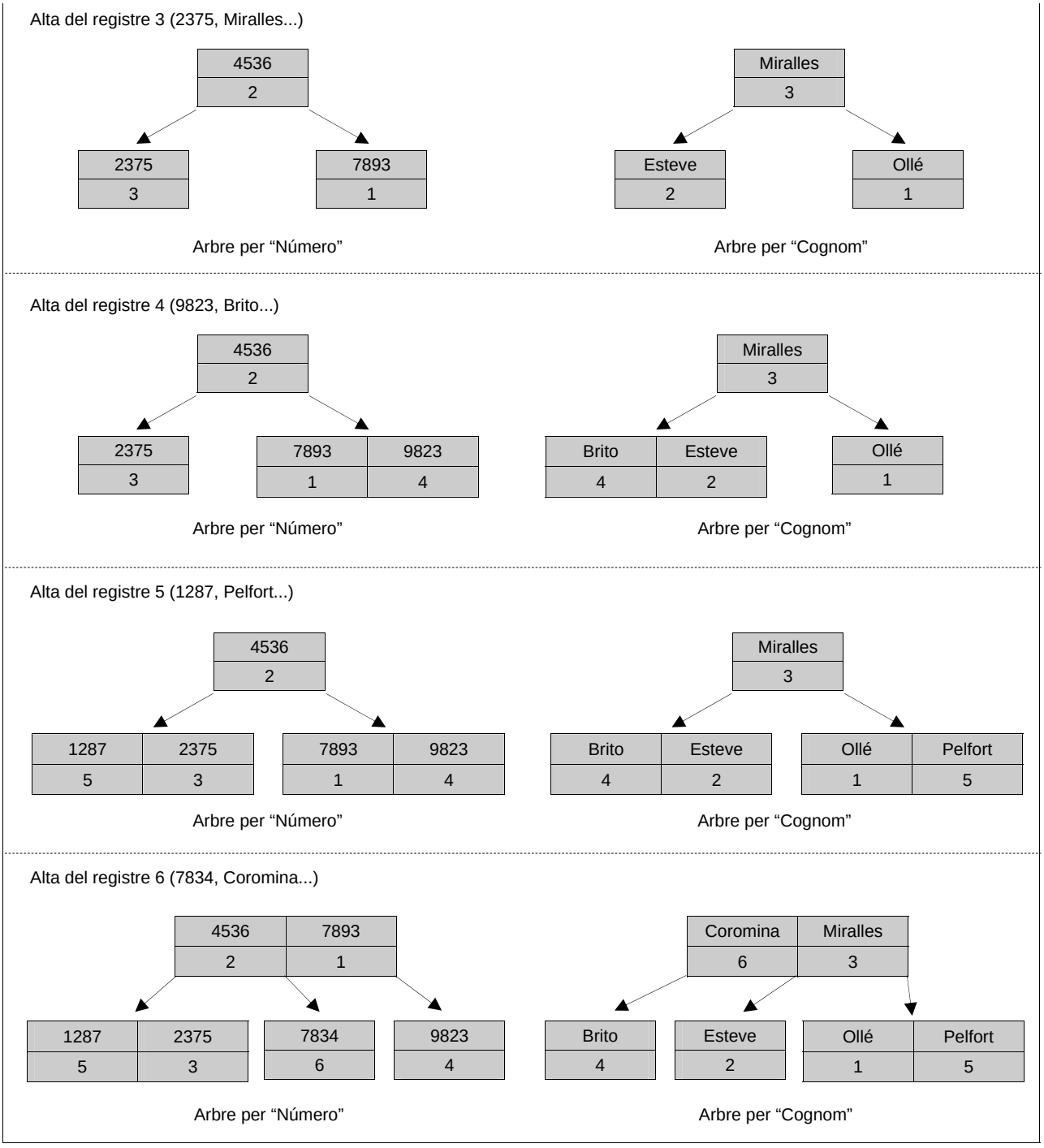
	Índex per "Número"	
	Número	Punter
1	1287	5
2	2375	3
3	4536	2
4	7834	6
5	7893	1
6	9823	4

	Índex per "Cognom"	
	Cognom	Punter
1	Brito	4
2	Coromina	6
3	Esteve	2
4	Miralles	3
5	Ollé	1
6	Pelfort	5

En l'exemple s'utilitza un índex lineal simple. Podríem, però, utilitzar els arbres B per als índexs, de manera que l'element a emmagatzemar a les 2d caselles de cada node sigui la parella formada (clau, punter). Així, si suposem arbres B d'ordre 1 i que es produeix la successió d'insercions en el mateix ordre que tenim els registres en l'arxiu de dades de la figura 21, podem anar construint els dos arbres B (un per a cada índex). La figura 22 ens mostra els passos que es succeirien.

Figura 22. Construcció d'arbres B per a índexs d'un arxiu de dades





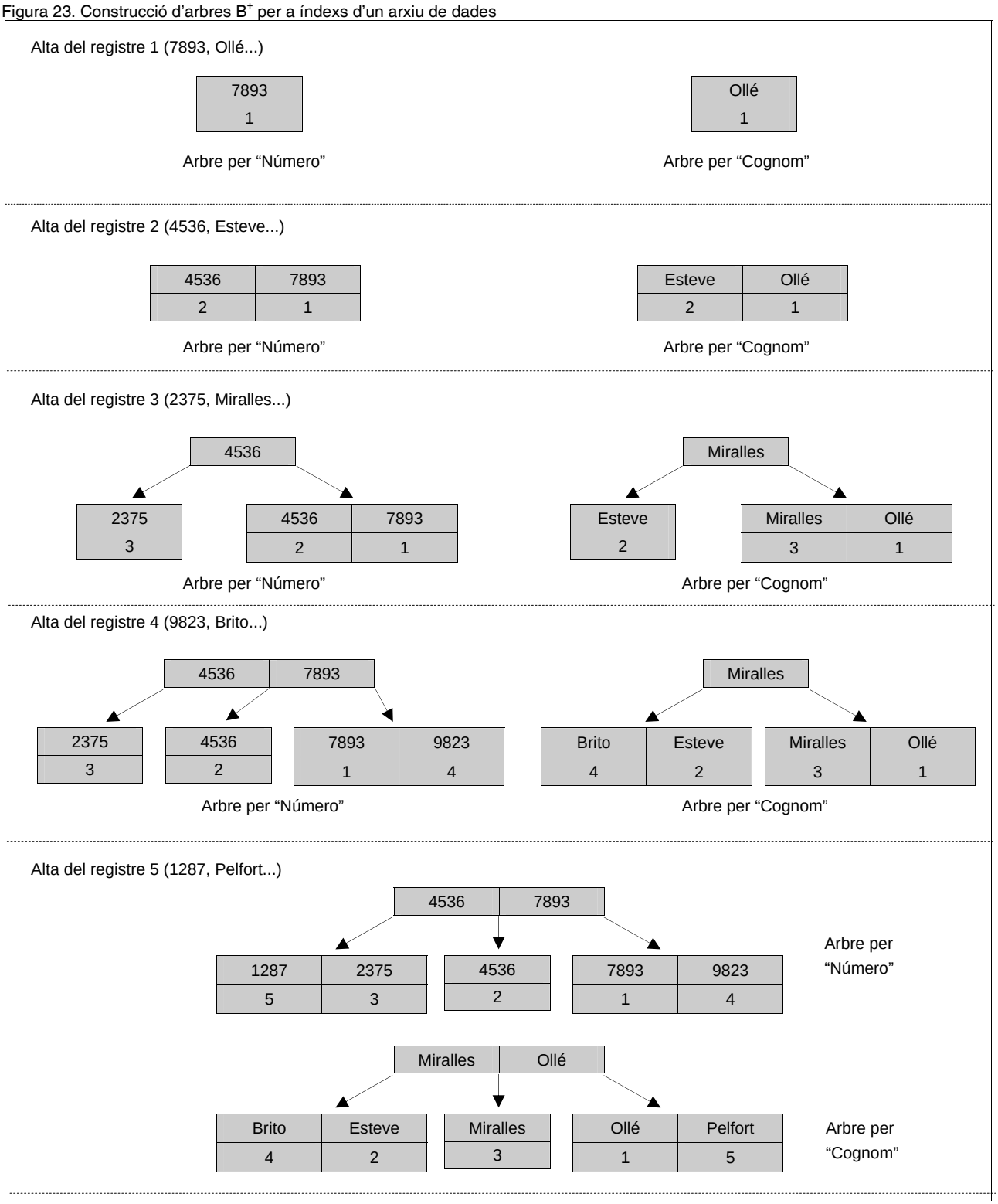
Observeu que la instrucció `llegir=_fi`, facilitada pel sistema gestor d'arxius indexat, per a cercar el registre amb cognom Coromina per a l'índex `cognom` aplicarà l'algorisme de recerca per a aquest índex. Trobarà el valor Coromina en el node arrel acompanyat del valor `6`, que indica l'adreça on es troba el corresponent registre en l'arxiu (o zona) de dades.

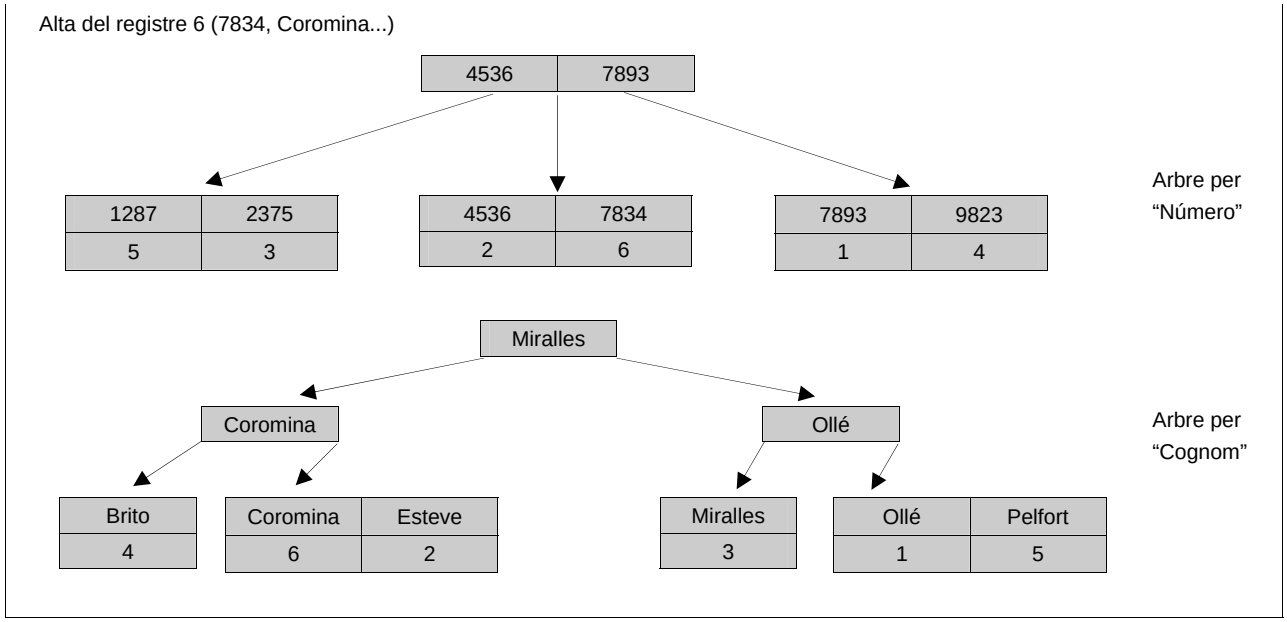
Fixeu-vos que per a aconseguir l'accés seqüencial per valor segons una via d'accés simplement cal dissenyar un algorisme d'inordre generalitzat per als diferents elements de cada node. Així, a partir de Coromina, passar al següent implicaria continuar pel camí que indica el seu punter

dret. Aniríem així a Esteve. El recorregut preordre generalitzat ens portaria, posteriorment, a Miralles, Ollé i Pelfort.

Refem els passos de la figura 22 pel cas que els índex siguin arbres B^+ d'ordre 1. La figura 23 ens mostra la seqüència de passos.

Figura 23. Construcció d'arbres B^+ per a índexs d'un arxiu de dades

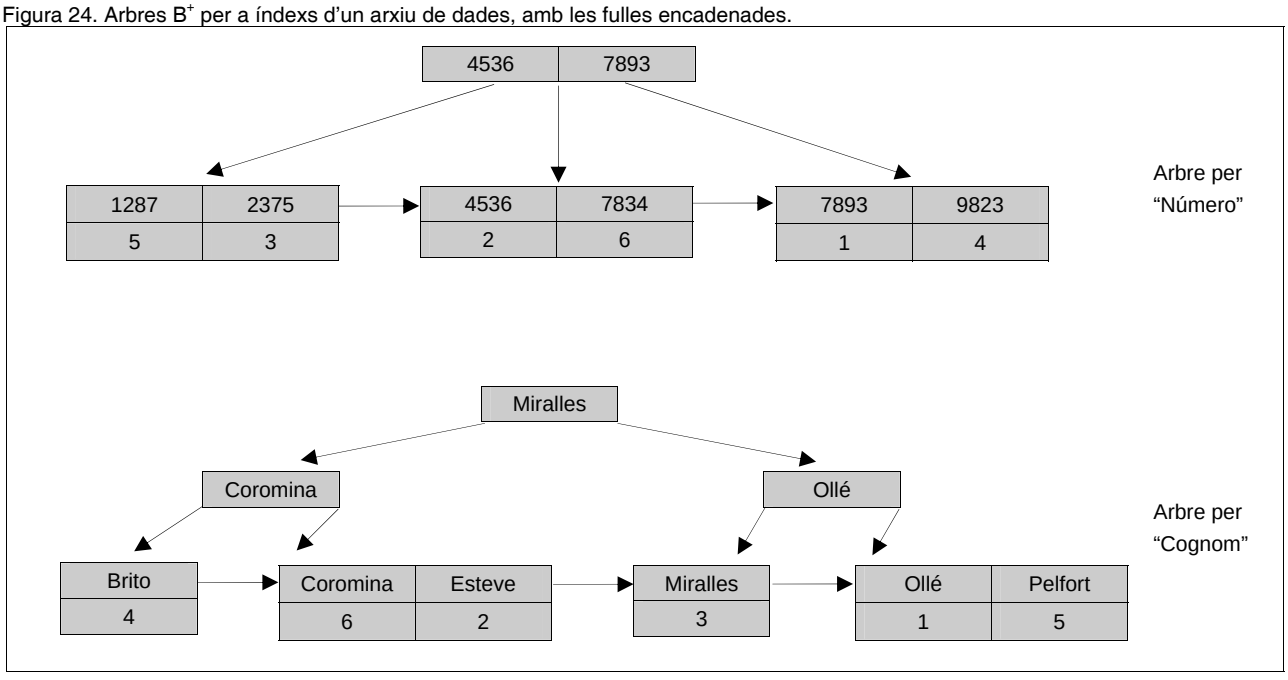




Observeu que en els nodes interiors dels arbres B^+ només hi ha el valor clau (valor de classificació), mentre que a les fulles és on es troben el valor clau i la direcció, on hi ha la posició del registre corresponent a la zona de dades.

Fixeu-vos que en cas de mantenir els nodes fulla encadenats (cosa independent de la gestió d'un arbre B^+), l'accés seqüencial pel valor de la via d'accés corresponent seria molt fàcil, ja que ens trobem Brito, Coromina, Esteve, Miralles, Ollé i Pelfort. Si en lloc d'una cadena simple es manté una doble cadena, tindríem el mateix accés seqüencial per valor cap endavant i cap enrere. La figura 24 mostra com podríem tenir els arbres obtinguts al final dels passos mostrats en la figura 23 en cas de mantenir encadenats els nodes de les fulles.

Figura 24. Arbres B^+ per a índexs d'un arxiu de dades, amb les fulles encadenades.



- Situem-nos en l'arbre que facilita la indexació dels registres per "número".

Suposem que s'acaba d'efectuar la instrucció de fitxers indexats (crèdit TF – PEM2) `llegir=_fi (f,r,"número", 4536)`, que vol dir la lectura directa per valor per la via d'accés "número" del valor 4536. El sistema gestor de fitxers, per processar aquesta lectura i facilitar el resultat, haurà efectuat la cerca per l'índex "número", arribant al node que conté el valor 4536, el qual va acompanyat del valor 2 que indica quin és el registre físic del fitxer on es troba el registre cercat.

Si després de l'execució d'aquesta instrucció, el programa executa la instrucció de fitxers indexats `llegirseg_fi (f,r,"número",clau)`, que vol dir la lectura seqüencial per valor per la via d'accés "número" a partir del registre situat actualment, el programa haurà de retornar el registre 6 (número 7834), cosa fàcil doncs està en el mateix node.

Però si després encara executa una altra instrucció `llegirseg_fi...` haurà de retornar el registre 1 (número 7893) i això li haurà estat molt senzill si tenim la cadena de punters en els nodes fulla. Això, en els arbres B, no és factible. Observeu la figura 22.

- Situem-nos en l'arbre que facilita la indexació dels registres per "cognom".

Suposem que s'acaba d'efectuar la instrucció de fitxers indexats (crèdit TF - PEM2) `llegir=_fi (f,r,"cognom", "Coromina")`, que vol dir la lectura directa per valor per la via d'accés "cognom" del valor "Coromina". El sistema gestor de fitxers, per processar aquesta lectura i facilitar el resultat, haurà efectuat la cerca per l'índex "cognom", arribant al node que conté el valor "Coromina", el qual va acompanyat del valor 6 que indica quin és el registre físic del fitxer on es troba el registre cercat.

Si després de l'execució d'aquesta instrucció, el programa executa la instrucció de fitxers indexats `llegirseg_fi (f,r,"cognom",clau)`, que vol dir la lectura seqüencial per valor per la via d'accés "cognom" a partir del registre situat actualment, el programa haurà de retornar el registre 2 (cognom "Esteve"), cosa fàcil doncs està en el mateix node.

Però si després encara executa una altra instrucció `llegirseg_fi...` haurà de retornar el registre 3 (cognom "Miralles") i això li haurà estat molt senzill si tenim la cadena de punters en els nodes fulla. Això, en els arbres B, no és factible. Observeu la figura 22.