Universidad Nacional Autónoma de Nicaragua – León

Facultad de Ciencias

Departamento de Computación

Trabajo Monográfico para optar al título de Licenciado en Computación

"Organización de Archivos de Base de Datos en Lenguaje C"

Integrantes:

Br. Heidy Paola Acevedo Rivas
Br. Marisol Argentina Pineda Blanco
Br. Lisseth de Carmen Soza García

Tutor:

MSc. Ricardo Espinoza Monterrey

León, Nicaragua. 30 de Septiembre del 2005

Indice de contenido

Indice de figuras

Indice de tablas

1. Introducción

Las técnicas de estructuración de datos aplicadas a conjuntos de datos, que los sistemas operativos manejan como "**cajas negras**", comúnmente se llaman **organización de archivos**. Un archivo tiene nombre, contenido, dirección donde se guarda y alguna información administrativa, por ejemplo, quién lo elaboró y cuán grande es.

En el presente documento se le presentan al estudiante los diferentes métodos de organización de archivos, impartidos en un capítulo de la asignatura de Base de Datos II, la cual se ofrece en el sexto semestre de la titulación **Ingeniería en Sistemas de Información**. Este capítulo se imparte aproximadamente en un período de un mes, equivalente a dieciséis horas.

Entre las diferentes técnicas de organización de archivos, se encuentran los archivos secuenciales, almacenamiento con diccionario de datos e indexación y asociación.

Este documento contiene información relativa a la implementación de los diferentes modelos de organización de archivos, programados en Lenguaje C, con el propósito de proporcionar al estudiante una guía de consulta sobre aplicaciones en un lenguaje de programación, de los conceptos básicos de organización de archivos.

2. Antecedentes

La asignatura Base de Datos, se ha venido impartiendo desde el inicio de la titulación Licenciatura en Computación, la cual nació en el año 1994, hasta la actual titulación Ingeniería en Sistemas de Información, la cual se originó en el año 2002.

Debido a la situación económica de nuestro país, la escasez de bibliografía en el Departamento de Computación ha sido un permanente problema para nuestros estudiantes. En particular, el área de Base de Datos ha sido una de las más golpeadas por dicha escasez, y se acentúa aún más, si tomamos en cuenta el rápido avance de las técnicas de gestión de bases de datos y la bibliografía que trae consigo este avance.

Hasta ahora, los estudiantes de Computación, han desarrollado, entre otros trabajos, temas monográficos que tienen que ver con la construcción de software de gestión de base de datos, basados en algún lenguaje de programación y un sistema gestor de base de datos. Pero hasta la fecha, no se cuenta con un documento en el que los estudiantes puedan consultar sobre un tema específico tanto a nivel teórico como práctico, de la asignatura Base de Datos.

De ahí nace la idea de elaborar este documento, en el que se exponen temas teóricos y prácticos de un capítulo específico de la asignatura Base de Datos II –Organización de Archivos-; como una alternativa para solucionar la problemática planteada anteriormente.

3. Justificación

La elaboración de este documento, se justifica en los siguientes puntos:

1. La necesidad de mostrar al estudiante que, los aspectos teóricos tratados en la asignatura Base de Datos II, con respecto al capítulo **Organización de Archivos**, pueden ser programados en algún lenguaje de programación específico, en este caso, Lenguaje C.

2. La inexistencia de un documento con información específica, de un tema de Base de Datos II, tratado tanto a nivel teórico como su construcción práctica en algún lenguaje de programación.

4. Objetivos

Generales

- Desarrollar ejercicios y programas del capítulo Organización de Archivos, de la asignatura Base de Datos II.

Específicos

- Elaborar programas en el lenguaje de programación C, para los ejercicios prácticos de la teoría relacionados con el capítulo Organización de Archivos.

- Organizar un manual de ayuda para los estudiantes de tercer año de Ingeniería en Sistemas de Información (ISI), con información teórica y ejercicios prácticos resueltos del capitulo de organización de archivos.

5. Metodología

La metodología a utilizar para el desarrollo de este documento, consiste en la organización (secuencia lógica) de programas de prácticas del capitulo Organización de Archivos. La asignatura consta de ocho horas totales por semana, repartidas en 4 horas teóricas y cuatro horas prácticas. Cada sesión de clase tiene una duración de dos horas, así que hay dos sesiones teóricas y dos sesión prácticas.

Cada subtema constará de:

- Una breve explicación teórica del subtema tratado.
- La realización completa de ejercicios ilustrativos del subtema.
- La inclusión de un programa en C, debidamente comentado, en el que se muestre el aspecto teórico tratado.

Todos los programas han sido escritos en Lenguaje C estándar, y compilados en **Visual C++ versión 6.0**. Se ha usado el entorno integrado de desarrollo Visual Studio 6.0.

La elaboración de este documento y los programas, se ha realizado en un equipo con las siguientes características hardware y software:

Procesador	Pentium III 700 MHz
Memoria RAM	128 MB
Capacidad de Disco Duro	20 GB
Sistema Operativo	Windows 2000 Professional con Service Pack 4.0
Aplicaciones de oficina	Microsoft Office 2000 Professional
Entorno Integrado de Desarrollo	Microsoft Visual Studio 6.0
Herramienta de diseño y dibujo	Microsoft Visio 2000

6. Organización de Archivos

La técnica utilizada para representar y almacenar registros en archivos es llamada organización de archivos. Los archivos se organizan lógicamente como secuencias de registros. Estos registros se corresponden con los bloques del disco.

Hay que tomar en consideración diversas maneras de representar los modelos lógicos de datos en términos de archivos. Aunque los bloques son de tamaño fijo determinado por las propiedades físicas del disco, los tamaños de los registros varían. En las base de datos relacionales las tuplas de las diferentes relaciones suelen ser de tamaños distintos.

Un enfoque de la correspondencia entre la base de datos y los archivos es utilizar varios y guardar los registros de cada una de las diferentes longitudes fijas existentes en cada uno de esos archivos. Los archivos con registros de longitud fija son más sencillos de implementar que los registros de longitud variable. Muchas de las técnicas utilizadas para los primeros pueden aplicarse al caso de longitud variable. Por lo tanto, se comienza considerando un archivo con registros de longitud fija.

6.1 Archivos con registros de longitud fija

Ejemplo: Considérese un archivo con registros de cuentas de la base de datos bancaria. Cada registro de este archivo se define de la manera siguiente:

```
Type deposito = record
        Numero_cuenta : char(10);
        Nombre_sucursal : integer;
        Saldo : real;
End
```

En este ejemplo el numero de bytes por registro es: 52 (integer 4 bytes, real 8 bytes, cada carácter 1 byte).

En la **Tabla** 6-1 se muestra un archivo que contiene los registros de cuentas para un banco.

Tabla 6-1: Archivo de registro de cuentas

	No_Cuenta	Nom_Suc	Saldo
registro 0	C-102	Chichigalpa	400
registro 1	C-305	León	350
registro 2	C-215	Chinandega	700
registro 3	C-101	Managua	500
registro 4	C-222	Matagalpa	700
registro 5	C-201	Jinotega	900
registro 6	C-217	Boaco	750
registro 7	C-110	Rivas	600
registro 8	C-218	Chontales	700
	4 bytes	40 bytes	8 bytes

Problemática con este enfoque sencillo:

- Resulta difícil borrar un registro de esta estructura. Se debe rellenar el espacio ocupado por el registro que hay que borrar con algún otro registro del archivo o tener algún medio de marcar los registros borrados para que puedan pasarse por alto.

- A menos que el tamaño de los bloques sea múltiplo de 40 (lo que resulta improbable) algunos de los registros se saltarán los límites de los bloques. Es decir, parte del registro se guardará en un bloque y parte en otro. Harán falta, por tanto, dos accesos a bloque para leer o escribir ese tipo de registro.

Posibles soluciones para garantizar la continuidad.

a) **Reordenamiento de los registros en el disco:** consiste en desplazar todos los registros hacia adelante de forma que el espacio que quedó libre con el borrado sea llenado.

Ejemplo: Supongamos los siguientes registros (los mismos que los de la **Tabla 6-1**):

	No_Cuenta	Nom_Suc	Saldo
registro 0	C-102	Chichigalpa	400
registro 1	C-305	León	350
registro 2	C-215	Chinandega	700
registro 3	C-101	Managua	500
registro 4	C-222	Matagalpa	700
registro 5	C-201	Jinotega	900
registro 6	C-217	Boaco	750
registro 7	C-110	Rivas	600
registro 8	C-218	Chontales	700

Borrando el registro 2, la secuencia quedaría como muestra la **Tabla** 6-2:

Tabla 6-2: Reordenando los registros después del borrado

	No_Cuenta	Nom_Suc	Saldo
registro 0	C-102	Chichigalpa	400
registro 1	C-305	León	350
registro 3	C-101	Managua	500
registro 4	C-222	Matagalpa	700
registro 5	C-201	Jinotega	900
registro 6	C-217	Boaco	750
registro 7	C-110	Rivas	600
registro 8	C-218	Chontales	700

Codificación en Lenguaje C de la técnica Reordenamiento de los registros en el disco

A continuación, veremos una propuesta de programa en Lenguaje C, que implementa la técnica anteriormente descrita.

Usaremos la siguiente estructura para manipular los registros:

```
typedef struct datos elemento;  /* declaración del tipo elemento */
struct datos                    /* elemento de una lista de enteros */
{
      int dato;
      elemento *siguiente;
};
```

La idea será llevar una lista lineal enlazada; cada elemento de la lista será un elemento de tipo **struct datos**.

Las inserciones se harán ordenadamente, las supresiones o eliminaciones se harán en cualquier parte de la lista, y luego del borrado se reorganizarán todos los registros de la lista.

Al ejecutar la aplicación, se mostrará un menú como el mostrado en la **Ilustración 6-1** .

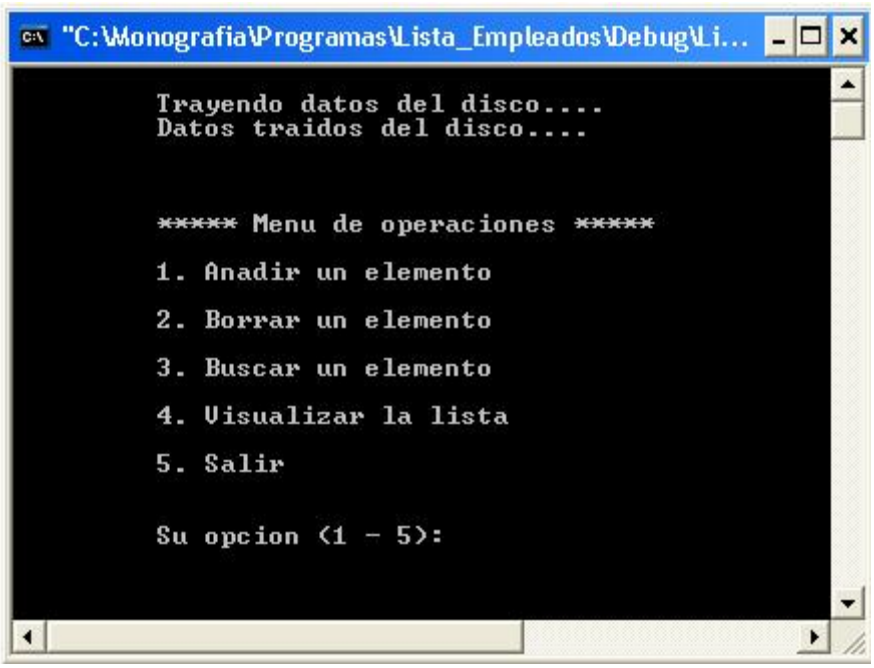


Ilustración 6-1: Menú de operaciones

A continuación se explicará cada una de las opciones del menú:

Opción 1

Al teclear dicha opción, se pedirá introducir un nuevo elemento tipo entero. En la **Ilustración 6-2** se muestran los elementos introducidos en ese instante.

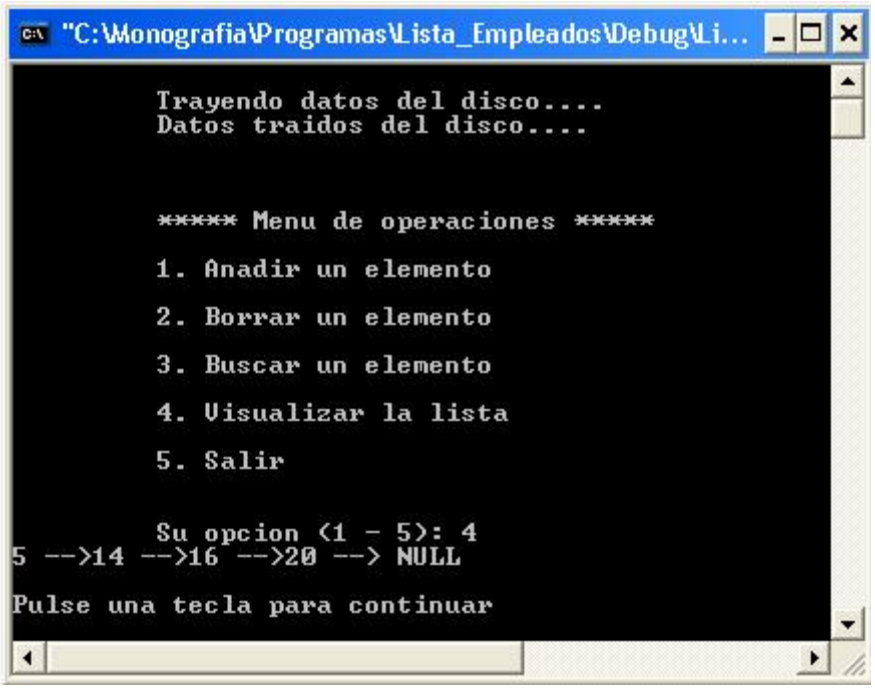


Ilustración 6-2: Elementos que contiene la lista

En la **Ilustración 6-3** , se muestra la introducción del elemento 3 en la lista.

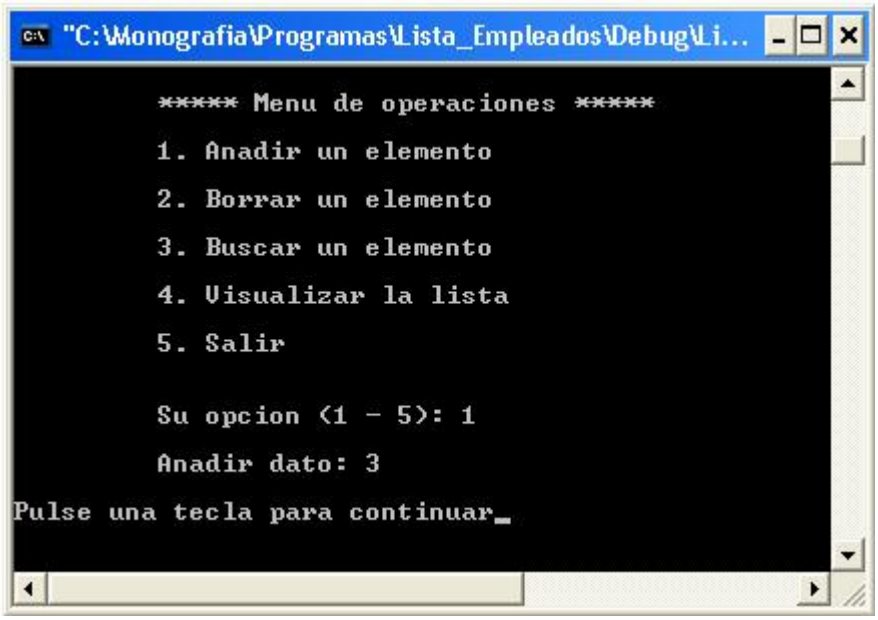


Ilustración 6-3: Añadiendo un elemento a la lista

La **Ilustración 6-4** muestra el estado de la lista después de introducir el elemento 3.

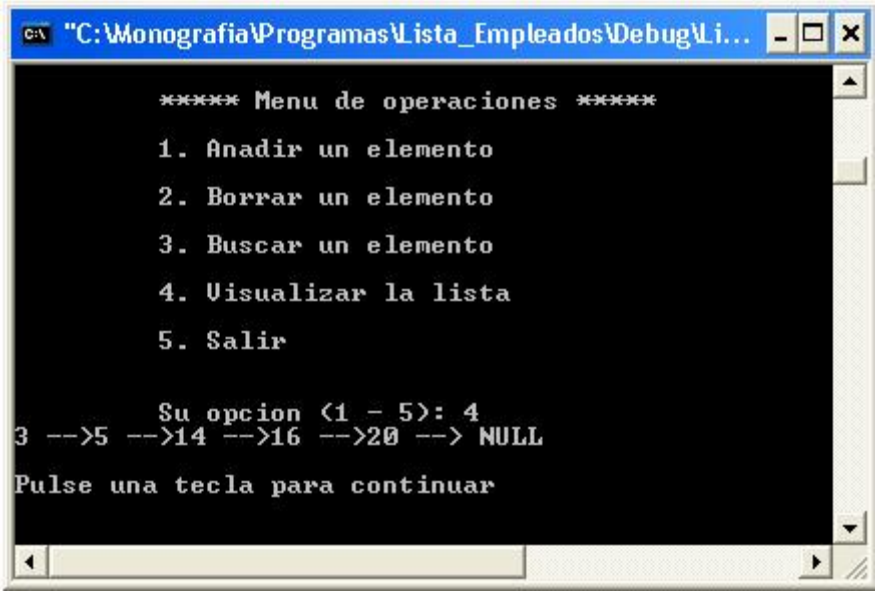


Ilustración 6-4: Estado de la lista después de introducir un elemento

Opción 2

Al teclear dicha opción se pedirá introducir el elemento a eliminar. La **Ilustración 6-5** , muestra la pantalla de borrado de un elemento.

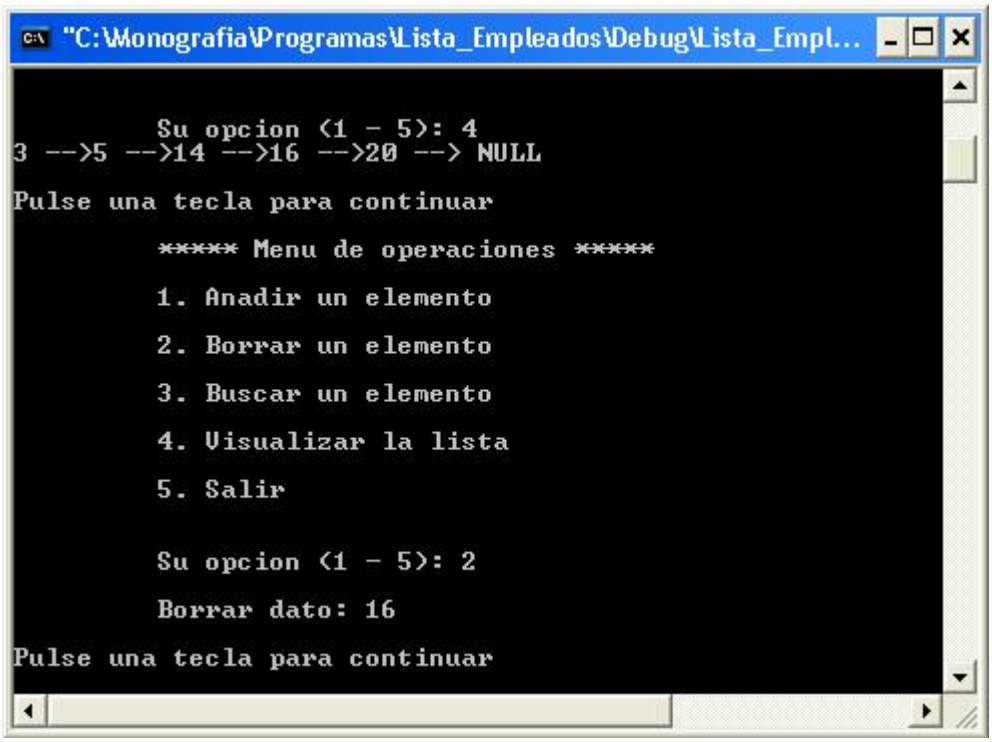


Ilustración 6-5: Borrado de un elemento

Luego del borrado del elemento 16, el estado de la lista queda como muestra la **Ilustración 6-6**.

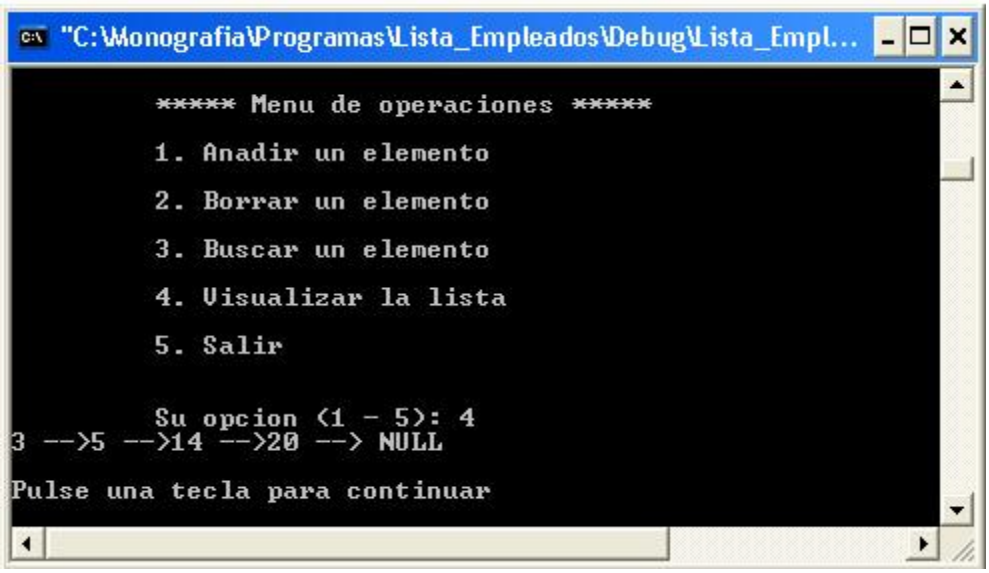


Ilustración 6-6: Visualización después del borrado

Opción 3

Esta opción permite buscar un elemento en la lista. La **Ilustración 6-7**, muestra la ejecución al teclear esta opción.

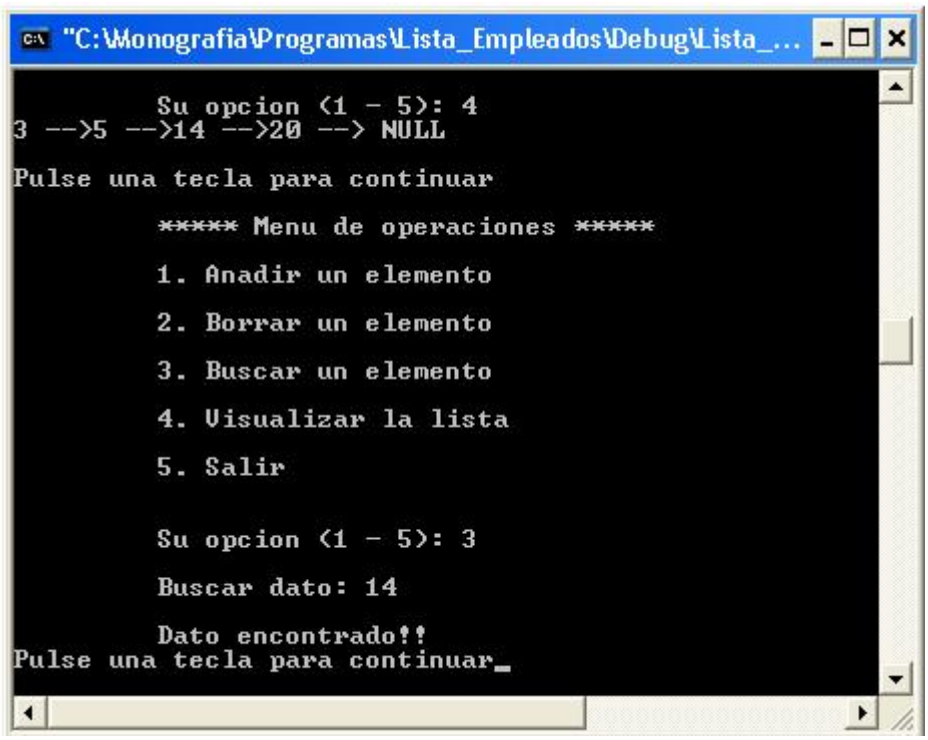


Ilustración 6-7: Búsqueda de un elemento

Opción 4
Esta opción visualiza los elementos de la lista. La **Ilustración 6-8**, muestra la ejecución de esta opción.

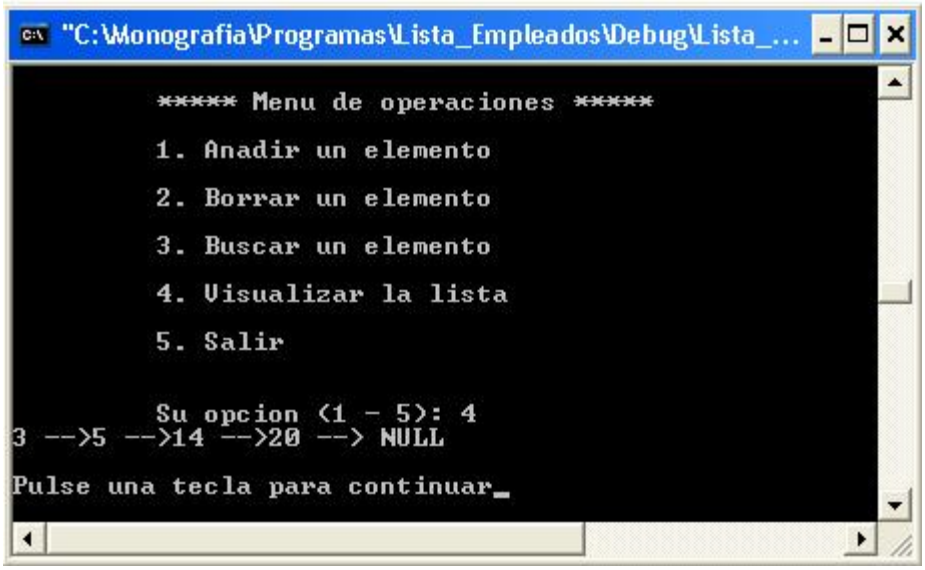


Ilustración 6-8: Visualización de la lista

Opción 5: Mediante esta opción, grabamos la lista al disco duro, y salimos del programa.

/* Este programa implementa un tipo de borrado que consiste en desplazar todos los registros hasta llenar la posición del elemento borrado
*/

//principal.cpp

```
/****************** Operaciones con listas **********************/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "Globales.h"

void main()                                          /* Función principal */
{
        FILE *pf;
        elemento *cabecera = NULL, *final = NULL;
        elemento *q;
        elemento aux;
        int opcion, dato;

        pf = fopen("datos", "rb");  // abrir el fichero datos para leer

        if (pf != NULL)
        {
          // El fichero existe, reconstruir la lista en MP
          // Traer datos del disco
          printf ("\n\t Trayendo datos del disco....");
          fread (&aux, tam_reg, 1, pf);

          while (!feof (pf) && !ferror (pf))
          {
                  IntroducirListaNueva (&cabecera, &final, aux.dato);
                  fread (&aux, tam_reg, 1, pf);
          }
          printf ("\n\t Datos traidos del disco....\n\n");

          fclose (pf);

        }


        while (1)
        {
          opcion = Menu();

          switch (opcion)
          {
                  case 1:
                          printf("\n\t Anadir dato: ");
                          scanf("%d",&dato);
                          Anadir (&cabecera,dato);
```

```
                    break;

            case 2:
                    printf("\n\t Borrar dato: ");
                    scanf ("%d",&dato);
                    Borrar (&cabecera,dato);
                    break;

            case 3:
                    printf("\n\t Buscar dato: ");
                    scanf("%d",&dato);
                    q = Buscar(cabecera,dato);

                    if(q)
                            printf ("\n\t Dato encontrado!!");
                    else
                            printf ("\n\t Lista vacia\n");
                    break;

            case 4:
                    Visualizar (cabecera);
                    break;

            case 5:  // Grabar en el disco y salir
                    printf ("\n\t Iniciada la copia de datos al disco...");
                    GrabarEnDisco (cabecera, pf);
                    printf ("\n\t Datos copiados al disco \n\n");

                    exit(0);

                    break;

          }
          printf("\nPulse una tecla para continuar");
          getch();
        }

}
```

// Globales.h

```
#define ListaVacia (cabecera==NULL)

/* Lista simplemente enlazada
   Cada elemento contiene un Nº entero
*/

typedef struct datos elemento;  /* declaración del tipo elemento */
struct datos                    /* elemento de una lista de enteros */
```

```
{
        int dato;
        elemento *siguiente;
};

int tam_reg = sizeof (elemento);   // tamanño de la estructura datos

/* Funciones prototipos */
elemento *NuevoElemento();
void Error();
int Menu();
void Anadir(elemento **, int);
void Borrar(elemento **, int);
elemento *Buscar(elemento *, int);
void Visualizar(elemento *);
void IntroducirListaNueva (elemento **, elemento**, int);
void GrabarEnDisco (elemento *, FILE *);


////////////////////////////////// Definición de Funciones ////////////////////////////////////////

/* Crear un nuevo elemento */
elemento *NuevoElemento()
{
        return ((elemento *)malloc(sizeof(elemento)));
}

/* Función Error */
void Error (void)
{
        perror ("Error: insuficiente espacio de memoria.\n");
        exit (1);
}

/* Función Menú */
int Menu()
{
        int op;

        do
        {
          printf("\n\n\t ***** Menu de operaciones ***** \n");
          printf("\n\t 1. Anadir un elemento\n");
          printf("\n\t 2. Borrar un elemento\n");
          printf("\n\t 3. Buscar un elemento\n");
          printf("\n\t 4. Visualizar la lista\n");
          printf("\n\t 5. Salir\n");
          printf("\n\n\t Su opcion (1 - 5): ");
          scanf ("%d", &op);
```

```
        }while ( op < 1 || op > 5);

        return (op);

}

/* Introducir un elemento ordenadamente en la lista */
void Anadir(elemento **cab, int dato)
{
        elemento *cabecera = *cab;
        elemento *actual = cabecera,*anterior = cabecera, *q;

        if (ListaVacia)             /*si está vacía, crear un nuevo elemento*/
        {
          cabecera = NuevoElemento();
          cabecera->dato = dato;
          cabecera->siguiente = NULL;
          *cab=cabecera;
          return;
        }

        /*Entrar en la lista y encontrar el punto de inserción*/
        while(actual != NULL && dato > actual->dato)
        {
          anterior = actual;
          actual = actual->siguiente;
        }

        /*Dos casos:
         *1) Insertar al principio de la lista
         *2) Insertar después de anterior (incluye insertar al final)
         */

        q = NuevoElemento();          /* se genera un nuevo elemento */
        if(anterior == actual)        /* insertar al principio */
        {
          q->dato = dato;
          q->siguiente = cabecera;
          cabecera = q;
        }

        else                                  /* insertar después de anterior */
        {
          q->dato = dato;
          q->siguiente = actual;
          anterior->siguiente = q;
        }
        *cab = cabecera;
}
```

```
/* Encontrar un dato y Borrarlo */
void Borrar(elemento **cab, int dato)
{
	elemento *cabecera = *cab;
	elemento *actual = cabecera, *anterior = cabecera;
	if (ListaVacia)
	{
	  printf("Lista Vacia\n");
	  return;
	}

	/* Entrar en la lista y encontrar el elemento a Borrar */
	while(actual !=NULL && dato != actual->dato)
	{
	  anterior = actual;
	  actual = actual->siguiente;
	}

	/* Si el dato no se encuentra retornar */

	if(actual == NULL)
	    return;

	/* Si el dato se encuentra. Borrar el elemento. */
	if(anterior == actual)			/* Borrar el elemento de cabecera. */
	  cabecera = cabecera->siguiente;

	else
	  anterior->siguiente = actual->siguiente;

	free(actual);
	*cab = cabecera;
}

/* Buscar un elemento determinado en la lista */
elemento *Buscar(elemento *cabecera, int dato)
{
	elemento *actual = cabecera;

	while(actual != NULL && dato != actual->dato)
	  actual = actual->siguiente;
	return (actual);
}

/* Visualizar la lista */
void Visualizar(elemento *cabecera)
{
	elemento *actual = cabecera;
```

```
        if (ListaVacia)
                printf("Lista Vacia\n");
        else
        {
          while (actual != NULL)
          {
                printf("%d -->",actual->dato);
                actual = actual->siguiente;
          }
          printf (" NULL");
          printf("\n");
        }
}

/* Grabar la lista en el disco */
void GrabarEnDisco (elemento *cab, FILE *puntero_fich)
{
        elemento *aux;

        puntero_fich = fopen("datos", "wb"); // Abrir el fichero datos para escribir

        // rewind (puntero_fich);

        aux = cab;  // aux apunta a la cabecera de la lista

        // Verificar si la lista está vacía
        if (aux == NULL)
        {
          printf ("\n\t LISTA VACIA!!!");
          return;
        }

        // La lista tiene al menos un elemento

        // Recorrer la lista y grabar en el fichero referenciado por puntero_fich
        // cada elemento de la lista
        while (aux != NULL)
        {
          fwrite (aux, tam_reg, 1, puntero_fich);
          aux = aux->siguiente;
        }

        // Cerrar el archivo
        fclose (puntero_fich);

} // fin de GrabarEnDisco()
```

```
// Introducir un elemento en la lista nueva
void IntroducirListaNueva (elemento **p, elemento **f, int dato)
{
	elemento *pc, *fc, *q;

	pc = *p;  // principio de la cola
	fc = *f;  // final de la cola

	q = NuevoElemento ();
	q->dato = dato;

	q->siguiente = NULL;

	if (fc == NULL)
	  pc = fc = q;

	else
	{
	  fc->siguiente = q;
	  fc = fc->siguiente;
	}

	*p = pc;
	*f = fc;
}
```

Algoritmos de las funciones más importantes

Algoritmo de la función Anadir

```
Function Anadir(elemento **cab, int dato) void
var cabecera, actual anterior, q: pointer to elemento
cabecera ← cab
actual ← cabecera
anterior ← cabecera

1. Si la lista no contiene elementos, entonces
      cabecera ← crear un nuevo elemento
      cabecera->dato ← dato
      cabecera->siguiente ← NULL
      actualizar cab
FinSi

   [Si la lista contiene al menos un elemento, entonces]
2. Si anterior ← actual
      [Encontrar la ubicación correspondiente del dato a insertar]
      Mientras actual <> NULL y dato > actual->dato
        anterior ←actual
        actual ←actual->siguiente
      FinMientras

      q ← crear un nuevo elemento
      [Si dato a insertar es el primero en la lista, entonces]
      Si anterior = actual, entonces
        q->dato ← dato
        q->siguiente ← cabecera
        cabecera ← q
      FinSi

3. SiNo
   [El dato a insertar no es el primero: después de anterior y antes de actual
   incluye insertar al final]
        q->dato ← dato
        q->siguiente ← actual
        anterior->siguiente ← q
      FinSiNo

      Actualizar cab
FinFunction
```

Algoritmo de la función Borrar

```
Function Borrar(elemento **cab, int dato) void
var cabecera, actual anterior, q: pointer to elemento
cabecera ← cab
actual ← cabecera
anterior ← cabecera

1. Si la lista está vacía, entonces
      Imprimir “Lista vacía”
      Regresar
FinSi

[Entrar en la lista y encontrar el elemento a Borrar]
2. Mientras actual <> NULL y dato <> actual->dato
      anterior ← actual
      actual ← actual->siguiente
FinMientras

3. Si actual = NULL, entonces
      Imprimir “Elemento no encontrado”
      Regresar
FinSi

4. Si anterior = actual, entonces
      cabecera ← cabecera->siguiente
FinSi

5. SiNo
      anterior->siguiente ← actual->siguiente
FinSiNo

6. Liberar actual
7. Actualizar cabecera
FinFunction
```

Algoritmo de la función GrabarEnDisco

```
Function GrabarEnDisco (elemento *cab, FILE *puntero_fich) void
var  aux: pointer to elemento
var  puntero_fich: pointer to FILE

1. Abrir el fichero y dejarlo referenciado por puntero_fich
2. aux ← cab
[Verificar si la lista está vacía]
3. Si aux = NULL, entonces
      Imprimir “Lista Vacía”
      Retornar
Fin Si
```

[Recorrer la lista y guardar cada elemento en el disco]
4. Mientras aux <> NULL
    Escribir el elemento referenciado por aux, al disco
    aux ← aux->siguiente
FinMientras

5.Cerrar el fichero

FinFunction

b) **Que el último ocupe el espacio dejado:** consiste en pasar el último registro al espacio que estaba ocupado por el registro eliminado.

Ejemplo:
Borrando el registro 2 y desplazando el último registro al espacio ocupado por el registro borrado, la secuencia quedaría como muestra la **Tabla** 6-3:

Tabla 6-3: Moviendo el último al espacio dejado por el borrado

	No_Cuenta	Nom_Suc	Saldo
registro 0	C-102	Chichigalpa	400
registro 1	C-305	León	350
registro 8	C-218	Chontales	700
registro 3	C-101	Managua	500
registro 4	C-222	Matagalpa	700
registro 5	C-201	Jinotega	900
registro 6	C-217	Boaco	750
registro 7	C-110	Rivas	600

Codificación en Lenguaje C de la técnica que el último elemento ocupe el espacio dejado

A continuación, veremos una propuesta de programa en Lenguaje C, que implementa la técnica anteriormente descrita.

Usaremos la siguiente estructura para manipular los registros:

```
typedef struct datos elemento; /*declaración del tipo elemento*/
struct datos                   /*elemento de una lista de enteros*/
{
      int dato;
      elemento *siguiente;
};
```

La idea será llevar una lista lineal enlazada; cada elemento de la lista será un elemento de tipo **struct datos**.

Las inserciones se harán al final de la lista, las supresiones o eliminaciones se harán en cualquier parte de la lista, y luego del borrado, el último elemento ocupará el espacio dejado por el registro eliminado.

La funcionalidad de este programa, es bastante similar al programa mostrado antes (reordenamiento de los registros), con excepción del borrado y la inserción. En este programa, los registros no se reorganizan al borrar un elemento, sino que, el espacio dejado por este, lo pasa a ocupar el último registro.

De acuerdo a lo expuesto, la ejecución de insertar un nuevo elemento se muestra en la **Ilustración 6-9** .

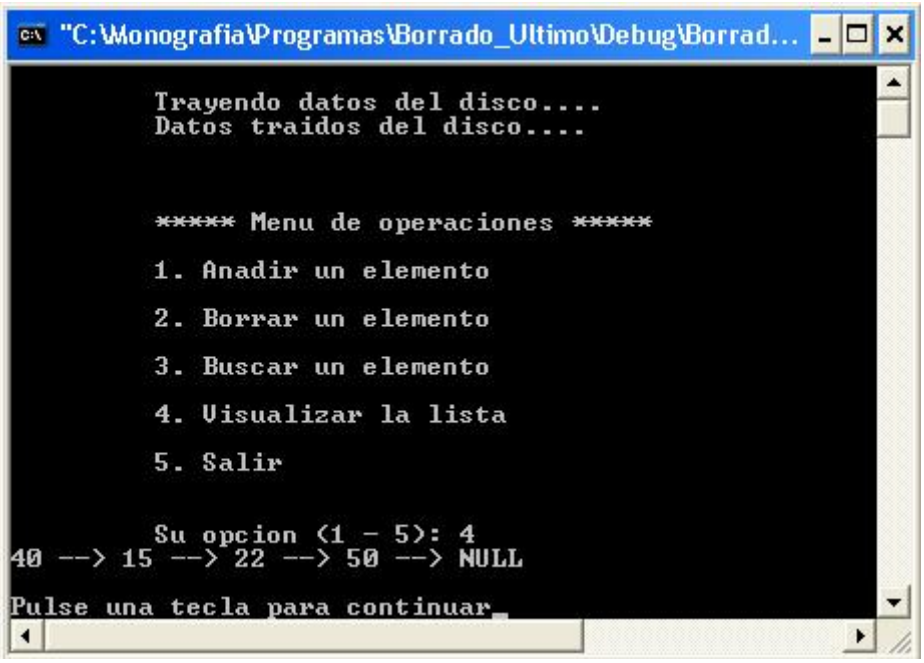


Ilustración 6-9: Estado de la lista antes de insertar un elemento

Opción1

Al insertar el elemento 10, el resultado es el que se muestra en la **Ilustración 6-10**.

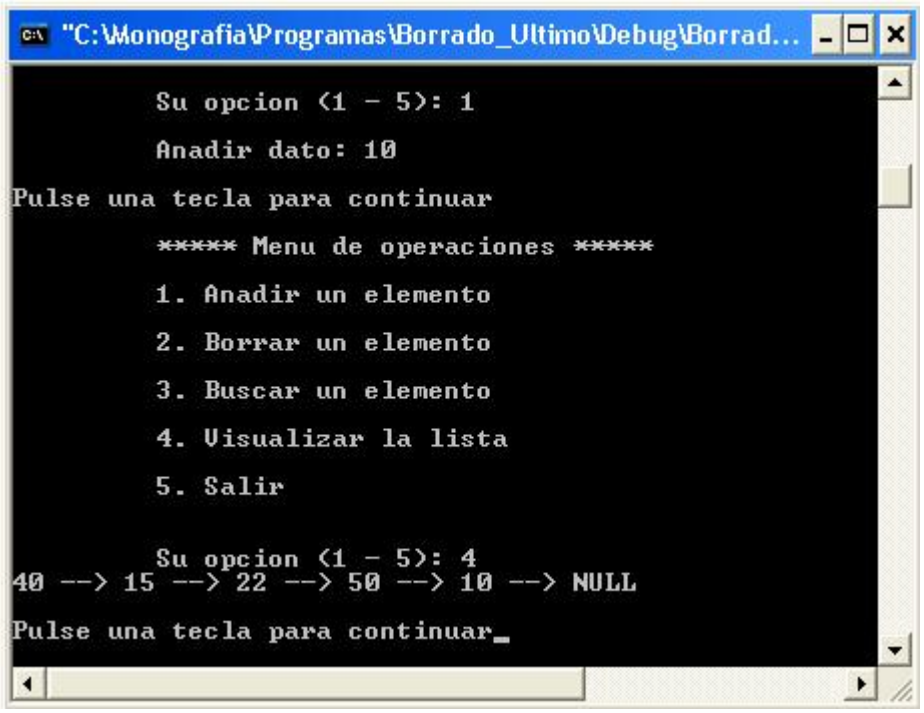


Ilustración 6-10: Inserción de un elemento al final de la lista

Opción 2

Esta opción permite eliminar un elemento de cualquier lugar de la lista, y el último registro pasa a ocupar el espacio dejado por el registro eliminado. En la **Ilustración 6-11**, se muestra un proceso de borrado.

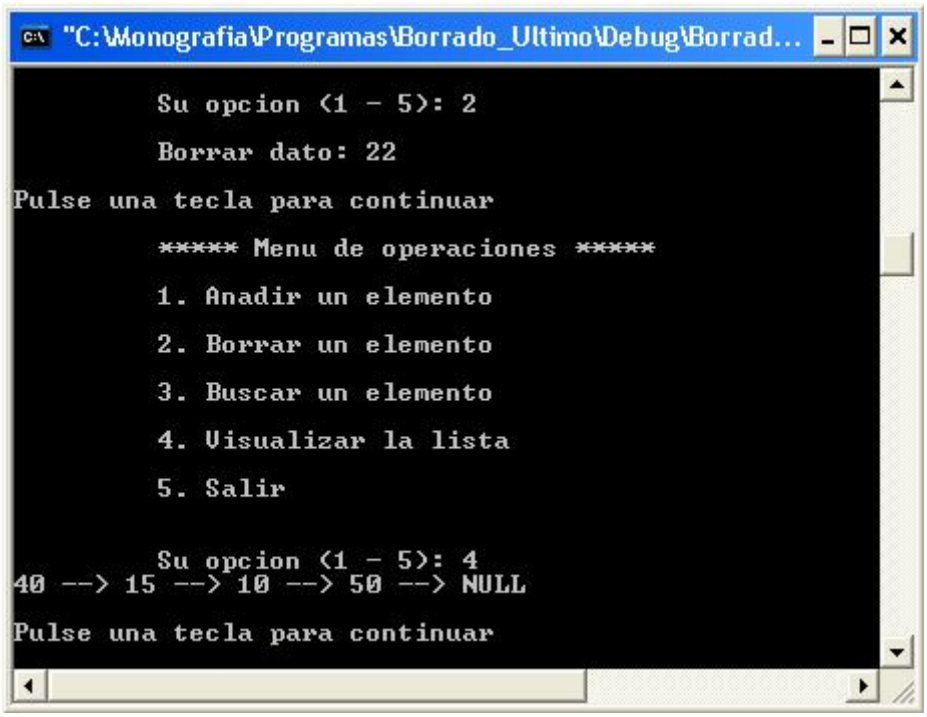


Ilustración 6-11: El último registro ocupa el espacio del elemento eliminado

En la **Ilustración 6-11**, se puede ver que el 10 (que era el último registro) pasó a ocupar la posición del elemento 22 (que fue eliminado).

```
/* Este programa implementa un tipo de borrado que consiste en mover el último
   elemento a la posición del elemento borrado
*/

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

void main()                                    /* Función principal */
{
      FILE *pf;
      elemento *cabecera = NULL, *final = NULL;
      elemento *q;
      elemento aux;
      int opcion, dato, k = 10;

      pf = fopen("datos", "rb");  // abrir el fichero datos para leer

      if (pf != NULL)
      {
```

```
  // El fichero existe, reconstruir la lista en MP
  // Traer datos del disco
  printf ("\n\t Trayendo datos del disco....");
  fread (&aux, tam_reg, 1, pf);

  while (!feof (pf) && !ferror (pf))
  {
        IntroducirListaNueva (&cabecera, &final, aux.dato);
        fread (&aux, tam_reg, 1, pf);
  }
  printf ("\n\t Datos traidos del disco....\n\n");

  fclose (pf);

}


while (1)
{

  opcion = Menu();

  switch(opcion)
  {
        case 1:
              printf ("\n\t Anadir dato: ");
              scanf ("%d",&dato);
              Anadir (&cabecera, &final, dato);
              break;

        case 2:
              if (cabecera == NULL)
              {
                    printf("\n\t LISTA VACIA!!!");
                    break;
              }

              printf ("\n\t Borrar dato: ");
              scanf ("%d",&dato);

              final = ApuntarUltimo (cabecera);
              Borrar (&cabecera, &final, dato);
              break;

        case 3:
              if (cabecera == NULL)
              {
                    printf("\n\t LISTA VACIA!!!");
                    break;
```

```
                    }

                    printf("\n\t Buscar dato: ");
                    scanf("%d",&dato);
                    q = Buscar(cabecera,dato);

                    if(q)
                        printf("\n\t Registro encontrado");
                    else
                        printf("\n\tLista vacia\n");
                    break;

                case 4:
                    if (cabecera == NULL)
                    {
                        printf("\n\t LISTA VACIA!!!");
                        break;
                    }

                    Visualizar (cabecera);
                    break;

                case 5:
                    printf ("\n\t Iniciada la copia de datos al disco...");
                    GrabarEnDisco (cabecera, pf);
                    printf ("\n\t Datos copiados al disco \n\n");

                    exit(0);
                    break;

            }
            printf("\nPulse una tecla para continuar");
            getch();
        }

}
```

/* globales.h */

```
/* definición de funciones y estructuras de datos */

#define ListaVacia (cabecera == NULL)

/*Lista simplemente enlazada
 *Cada elemento contiene un nº entero
 */

typedef struct datos elemento; /*declaración del tipo elemento*/
struct datos                   /*elemento de una lista de enteros*/
{
        int dato;
        elemento *siguiente;
};

int tam_reg = sizeof (elemento);   // tamaño de la estrutura datos

/* Funciones prototipos */
elemento *NuevoElemento();
void Error();
int Menu();
void Anadir (elemento **, int);
void Borrar (elemento **, elemento **, int);
elemento *Buscar (elemento *, int);
void Visualizar (elemento *);
elemento *ApuntarUltimo (elemento *);  // Apuntar al último elemento
void IntroducirListaNueva (elemento **, elemento**, int);
void GrabarEnDisco (elemento *, FILE *);

////////////////////////////////// Definición de Funciones ////////////////////////////////////////
/* Crear un nuevo elemento */
elemento *NuevoElemento()
{
        return ((elemento *)malloc(sizeof(elemento)));
}

/* Función Error */
void Error (void)
{
        perror ("Error: insuficiente espacio de memoria.\n");
        exit (1);
}

/* Función Menú */
int Menu()
{
        int op;
```

```
        do
        {
          printf("\n\n\t ***** Menu de operaciones ***** \n");
          printf("\n\t 1. Anadir un elemento\n");
          printf("\n\t 2. Borrar un elemento\n");
          printf("\n\t 3. Buscar un elemento\n");
          printf("\n\t 4. Visualizar la lista\n");
          printf("\n\t 5. Salir\n");
          printf("\n\n\t Su opcion (1 - 5): ");
          scanf ("%d", &op);
        }while ( op < 1 || op > 5);

        return (op);

}

/* Introducir un elemento ordenadamente en la lista */
void Anadir(elemento **p, elemento **f, int dato)
{
        elemento *pc, *fc, *q;

        pc = *p;
        fc = *f;

        q = NuevoElemento();
        q->dato = dato;
        q->siguiente = NULL;

        if (fc == NULL)
          pc = fc = q;

        else
          fc = fc->siguiente = q;

        *p = pc;
        *f = fc;

}

/* Encontrar un dato y Borrarlo */
void Borrar (elemento **cab, elemento **final, int dato)
{
        elemento *cabecera = *cab;
        elemento *fin = *final;
        elemento *actual = cabecera, *anterior = cabecera;

        if (ListaVacia)
        {
          printf("Lista Vacia\n");
```

```
  return;
}

/*entrar en la lista y encontrar el elemento a Borrar*/
while (actual != NULL && dato != actual->dato)
{
  anterior = actual;
  actual = actual->siguiente;
}

/*si el dato no se encuentra retornar*/

if(actual == NULL)
    return;

/* Si el dato se encuentra. Borrar el elemento. */
if(anterior == actual)    /* Borrar el elemento de cabecera. */
{
  // Verificar si la lista tiene un sólo elemento
  if (anterior->siguiente == NULL)
  {
          free (*cab);
          *cab = NULL;
          *final = NULL;
  }

  else    // La lista tiene más de un elemento
  {
          // Copiar el último elemento al elemento a borrar
          anterior->dato = fin->dato;

          // Liberar la memoria que ocupa el último elemento
          while (actual->siguiente != fin)
                  actual = actual->siguiente;

          actual->siguiente = NULL;
          free (fin);

          *cab = cabecera;
          *final = actual;

  }

}

// Borrar un elemento no cabecera. Incluye borrar al final
else
{
  // Ver si el ultimo
```

```
        if (actual->siguiente == NULL)
        {
                anterior->siguiente = actual->siguiente;
                free (actual);
                *cab = cabecera;
                *final = anterior;
        }


        else
        {
                // Borrar un elemento del medio de la lista
                actual->dato  = fin->dato;

                while (actual->siguiente != fin)
                        actual = actual->siguiente;

                actual->siguiente = NULL;
                free (fin);

                *cab = cabecera;
                *final = actual;
        }

      }

}


/* Buscar un elemento determinado en la lista */
elemento *Buscar(elemento *cabecera, int dato)
{
      elemento *actual = cabecera;

      while(actual != NULL && dato != actual->dato)
        actual = actual->siguiente;
      return (actual);
}

/* Visualizar la lista */
void Visualizar(elemento *cabecera)
{
      elemento *actual = cabecera;

      if (ListaVacia)
      {
        printf("Lista Vacia \n");
        return;
      }
```

```
        else
        {
          while(actual != NULL)
          {
                  printf("%d --> ",actual->dato);
                  actual = actual->siguiente;
          }

          printf ("NULL");
          printf("\n");
        }
}

/////// Devolver un puntero al último elemento  //////////
elemento *ApuntarUltimo (elemento *cabecera)
{
        elemento *caminante = cabecera;
        elemento *anterior = caminante;

        // Recorrer la lista y parar en el último

        if (cabecera == NULL)
          return (NULL);

        if (caminante->siguiente == NULL)  // La lista tiene un solo elemento
          return (caminante);

        // Recorrer la lista y dejar apuntado al último con anterior
        while (caminante != NULL)
        {
          anterior = caminante;
          caminante = caminante->siguiente;

        }

        return (anterior);

}

/* Grabar la lista en el disco */
void GrabarEnDisco (elemento *cab, FILE *puntero_fich)
{
        elemento *aux;

        puntero_fich = fopen("datos", "wb");

        // rewind (puntero_fich);

        aux = cab;  // aux apunta a la cabecera de la lista
```

```
        // Verificar si la lista está vacía
        if (aux == NULL)
        {
          printf ("\n\t LISTA VACIA!!!");
          return;
        }

        // La lista tiene al menos un elemento

        // Recorrer la lista y grabar en el fichero referenciado por puntero_fich
        // cada elemento de la lista
        while (aux != NULL)
        {
          fwrite (aux, tam_reg, 1, puntero_fich);
          aux = aux->siguiente;
        }

        // Cerrar el archivo
        fclose (puntero_fich);

} // fin de GrabarEnDisco()

// Introducir un elemento en la lista nueva
void IntroducirListaNueva (elemento **p, elemento **f, int dato)
{
        elemento *pc, *fc, *q;

        pc = *p;  // principio de la cola
        fc = *f;  // final de la cola

        q = NuevoElemento ();
        q->dato = dato;
        q->siguiente = NULL;

        if (fc == NULL)
          pc = fc = q;

        else
        {
          fc->siguiente = q;
          fc = fc->siguiente;
        }

        *p = pc;
        *f = fc;
}
```

Algoritmos de las funciones más importantes

Algoritmo de la función Anadir

```
Function Anadir(elemento **p, elemento **f, int dato) void
var pc, fc, q: pointer to elemento
pc ←p
pf ←f

q← crear un nuevo elemento
q->dato ← dato
q->siguiente ← NULL

[Si la lista está vacía, entonces, añadir el elemento]
1. Si fc = NULL, entonces
        pc ← fc ← q
FinSi

[La lista tiene al menos un elemento]
2. SiNo
        fc->siguiente ← q
        fc ← q
FinSiNo

[Actualizar cabecera y final de la lista]
p ← pc
f ← fc

FinFunction
```

Algoritmo de la función Borrar

```
Function Borrar (elemento **cab, elemento **final, int dato) void
var cabecera ←cab, fin ← final, actual ← cabecera, anterior ← cabecera

1 Si la lista está vacía, entonces
      Imprimir “Lista vacía”
      Regresar
FinSi

[Entrar en la lista y encontrar el elemento a Borrar]
2. Mientras actual <> NULL y dato <> actual->dato
      anterior ← actual
      actual ← actual->siguiente
FinMientras

[Si el dato no se encuentra retornar]
3. Si actual = NULL, entonces
      Imprimir “Elemento no encontrado”
      Regresar
FinSi

[Si el dato se encuentra. Borrar el elemento]
[Primer caso: borrar el elemento cabecera
4. Si anterior = actual, entonces
      [Verificar si la lista tiene un sólo elemento]
      Si anterior->siguiente = NULL
        Liberar cab
        cab ← NULL
        final ← NULL
      FinSi

      [La lista tiene más de un elemento]
      SiNo
        [Copiar el dato del último elemento al dato del elemento a borrar]
        anterior->dato ← fin->dato
        [Liberar la memoria que ocupa el último elemento]
        Mientras actual->siguiente <> fin
              actual ← actual->siguiente
        FinMientras
      actual->siguiente ← NULL
      Liberar fin
      cab ← cabecera
      final ← actual
      FinSiNo
FinSi
```

[**Segundo caso**: Borrar un elemento no cabecera. Incluye borrar al final]
SiNo
[ver si es el último]
Si actual->siguiente = NULL, entonces
anterior->siguiente ← actual->siguiente
Liberar actual
cab ← cabecera
final ← anterior
FinSi

SiNo
[Borrar un elemento del medio de la lista]
actual->dato ← fin->dato
Mientras actual->siguiente <> fin
actual ← actual->siguiente
FinMientras
actual->siguiente ← NULL
Liberar fin
cab ← cabecera
final ← actual
FinSiNo
FinSiNo

FinFunction

Algoritmo de la función GrabarEnDisco
Function GrabarEnDisco (elemento *cab, FILE *puntero_fich) void
var aux: pointer to elemento
var puntero_fich: pointer to FILE

1. Abrir el fichero y dejarlo referenciado por puntero_fich
2. aux ← cab
[Verificar si la lista está vacía]
3. Si aux = NULL, entonces
Imprimir “Lista Vacía”
Retornar
Fin Si

[Recorrer la lista y guardar cada elemento en el disco]
4. Mientras aux <> NULL
Escribir el elemento referenciado por aux, al disco
aux ← aux->siguiente
FinMientras

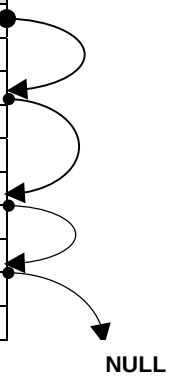
5.Cerrar el fichero
FinFunction

c) **Uso de punteros colgantes:** Se usa un registro cabecera, el cual va a contener la dirección del primer registro eliminado. Al ser ocupado el espacio éste apuntará al próximo vacío.

Ejemplo: Borrando los registros 1, 4 y 6 de la **Tabla** 6-1, la secuencia quedaría como muestra la **Tabla** 6-4.

Tabla 6-4: Usando punteros para marcar registros borrados

	No_Cuenta	Nom_Suc	Saldo
Cabecera			
Registro 0	C-102	Chichigalpa	400
Registro 1			
Registro 2	C-215	Chinandega	700
Registro 3	C-101	Managua	500
Registro 4			
Registro 5	C-201	Jinotega	900
Registro 6			
Registro 7	C-110	Rivas	600
Registro 8	C-218	Chontales	700



Codificación en Lenguaje C de la técnica del puntero colgante

A continuación, veremos una propuesta de programa en Lenguaje C, que implementa la técnica anteriormente descrita.

Usaremos la siguiente estructura para manipular los registros:

```
#define MAX_NUM_ELTOS 100    // Define el máximo número de estudiantes
typedef struct
{
      char carnet[20];         // Carnet del estudiante
      char nombre[50];         // Nombre del estudiante
      int edad;                // Edad del estudiante

}Info_Alumno;

struct Lista
{
      Info_Alumno Al;              // Información de un alumno
      struct Lista *sig_borrado;   // Puntero al siguiente registro eliminado
      char estado_reg;             // Estado del registro: 'b' --> borrado
                                   //                      'o' --> ocupado
                                   //                      'v' --> vacío
};
struct Lista LAlumnos[MAX_NUM_ELTOS];       // Array de alumnos
```

En la **Ilustración 6-12**, podemos ver la representación gráfica de la estructura de datos que usaremos para implementar la técnica.

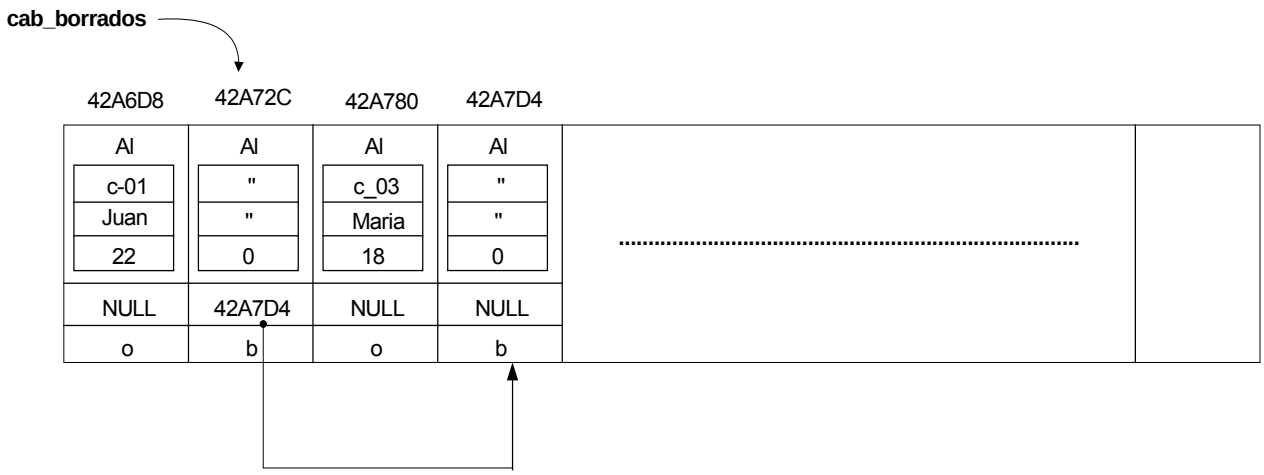


Ilustración 6-12: Estructura de datos en la técnica del puntero colgante.

Descripción de la estructura Info_Alumno

```
typedef struct
{
      char carnet[20];        // Carnet del estudiante
      char nombre[50];        // Nombre del estudiante
      int edad;               // Edad del estudiante

}Info_Alumno;
```

Campo	Descripción
carnet	Representa la identificación del estudiante
nombre	Representa el nombre del estudiante
edad	Representa la edad del estudiante

Descripción de la estructura Lista

```
struct Lista
{
      Info_Alumno Al;             // Información de un alumno
      struct Lista *sig_borrado;  // Puntero al siguiente registro eliminado
      char estado_reg;            // Estado del registro: 'b' --> borrado
                                  //                      'o' --> ocupado
                                  //                      'v' --> vacío
};
```

Campo	Descripción
Al	Representa la información del alumno
sig_borrado	Puntero al siguiente borrado
estado_reg	Representa el estado del registro: borrado, ocupado, vacío

La información de cada estudiante la controlaremos mediante un arreglo de dimensión MAX_NUM_ELTOS. Por eso, realizamos la siguiente declaración:

struct Lista LAlumnos[MAX_NUM_ELTOS]; // Array de alumnos

Explicación del funcionamiento de la técnica en nuestra estructura de datos

El arreglo LAlumnos, se inicializará a valores vacíos al iniciar el programa: la cadena vacía para las cadenas de caracteres, un valor de cero para los campos enteros, y el valor NULL para el puntero. Cada vez que se requiera introducir un elemento, se introducirá después del último, si antes del último, no hay registros borrados. Si hubiese al menos, un registro borrado, entonces al introducir un registro nuevo, se introducirá en la posición de dicho registro.

Ya que la idea general es llevar un control de los elementos que han sido borrados, existirá una cabecera llamada **cab_borrados**, que apuntará –si existe-, al primer elemento borrado en la lista. Esta cabecera no es más que un puntero a struct Lista, inicializado a NULL (no hay elementos borrados). La siguiente línea muestra cómo declarar dicho puntero:

struct Lista *cab_borrados = NULL;

Cuando se borra un primer elemento del arreglo, entonces, sus campos se ponen a valores vacíos (en el caso de la información de alumno, dada por el campo Al, dichos campos también se pondrán a valores vacíos: la cadena vacía, -“”-, para las cadenas de caracteres, y un valor de cero , 0, para el valor entero que representa la edad), el puntero **sig_borrado**, que inicialmente estaba a NULL, seguirá estando a NULL, mientras a continuación de él (en posiciones adelante) no hayan registros borrados.

Si después de borrado el primero, se borra otro elemento, entonces, el puntero sig_borrado del elemento anterior inmediato borrado, apuntará a este nuevo registro que acaba de ser borrado.

Por ejemplo, en la **Ilustración 6-12**, existen hasta ese momento, dos elementos borrados, el que ocupa la posición 1 y el otro que ocupa la posición 3 del arreglo. En dicha figura, se han enumerado sus posiciones de memoria en el arreglo. Así, **cab_borrados**, apunta inicialmente al elemento con índice 1, y a su vez, el puntero **sig_borrado** de este elemento, tiene el valor **42A7D4**, que es la posición del siguiente elemento borrado. Esta concatenación se sigue para los siguientes registros borrados que pudiese tener la lista.

A como se explicó anteriormente, si hubiese algún registro borrado, y queremos insertar un nuevo elemento a la lista, éste se insertará en la posición del primer elemento borrado, debiendo actualizar la cabecera **cab_borrados**, para que ahora, apunte al siguiente elemento borrado si existe, o a NULL si no hubiera ninguno más.

Igual que en las técnicas anteriores, al salir del programa, la lista completa se escribe a un fichero binario en el disco duro. Al correr de nuevo el programa, se reconstruye la lista en memoria principal y se trabaja con ella a como estaba en su estado inicial.

Funcionamiento de la aplicación

A continuación, veremos la funcionalidad que tiene la aplicación. Se explicará cada opción del menú.

El menú presentado al iniciar la aplicación es el que se muestra en la **Ilustración 6-13**:

Ilustración 6-13: Menú inicial en la técnica del puntero colgante

En la **Ilustración 6-13**, se puede ver, que al ejecutar la aplicación, ya existían datos en el fichero en el disco duro, por lo que se tiene que reconstruir la lista en memoria principal.

En la **Ilustración 6-14**, se puede ver el estado inicial de la lista, cuando se ha reconstruido en memoria principal.

```
Teclee su opcion (1-5): 3


        ALUMNO No 1
        -------------------------------
        Direccion del elemento 0: 42A6D8
        Carnet: c-01
        Nombre: Juan
        Edad: 22
        Puntero al sig. borrado: 0
        Estado del registro: o

        ALUMNO No 2
        -------------------------------
        Direccion del elemento 1: 42A72C
        Carnet:
        Nombre:
        Edad: 0
        Puntero al sig. borrado: 42A7D4
        Estado del registro: b

        ALUMNO No 3
        -------------------------------
        Direccion del elemento 2: 42A780
        Carnet: c-03
        Nombre: Maria
        Edad: 18
        Puntero al sig. borrado: 0
        Estado del registro: o

        ALUMNO No 4
        -------------------------------
        Direccion del elemento 3: 42A7D4
        Carnet:
        Nombre:
        Edad: 0
        Puntero al sig. borrado: 0
        Estado del registro: b

        cabecera_borrados apunta a: 42A72C

Presione una tecla para continuar . . .
```

Ilustración 6-14: Estado inicial de la lista cuando se ha traído del disco

Las entradas del usuario, están en negrita, cursiva y subrayadas. La **Ilustración 6-14**, muestra que al momento de ejecutar la aplicación, hay 2 registros eliminados de un total de 4 ingresados. Los estudiantes que han sido eliminados son el 2 y el 4. A como se puede observar, al final del listado, se imprime también el valor de **cab_borrados**, la cual contiene el valor **42A72C**, que es la dirección del elemento que representa al alumno 2.

Opción 1: Añadir un alumno a la lista

Esta opción añade un alumno a la lista. Si no hay elementos borrados, será añadido después del último. En cambio, si hay al menos, un elemento borrado, se insertará en la posición del primer elemento borrado, y a continuación, se actualizará la cabecera de borrados.

La **Ilustración 6-15**, muestra la interfaz que permite insertar un nuevo estudiante a la lista.

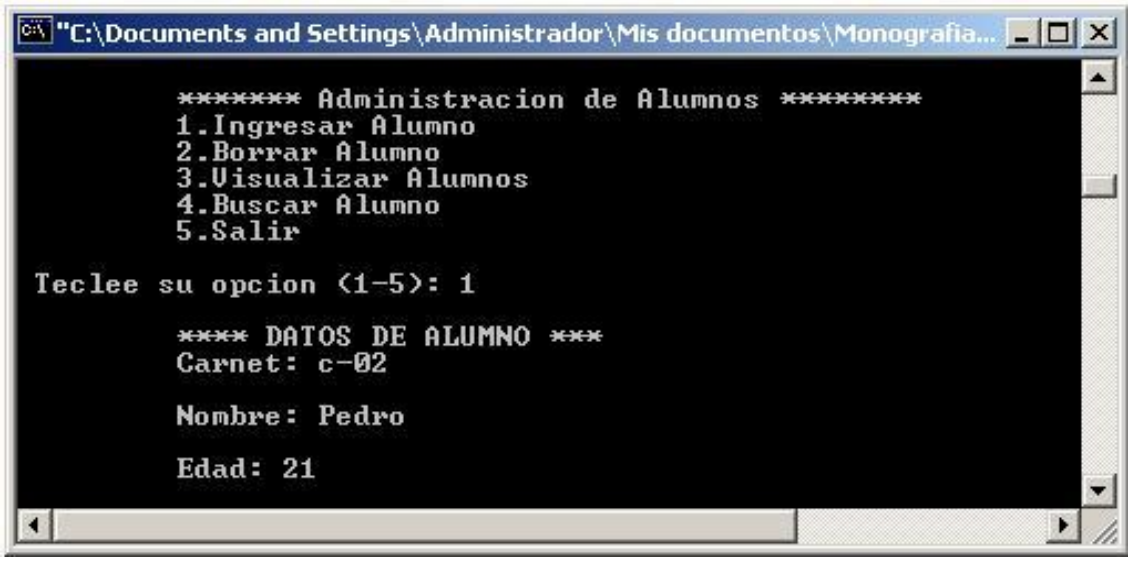


Ilustración 6-15: Inserción de un alumno – técnica de puntero colgante

La **Ilustración 6-16**, muestra el estado del array, una vez insertado al alumno Pedro.

```
******* Administracion de Alumnos *******
1.Ingresar Alumno
2.Borrar Alumno
3.Visualizar Alumnos
4.Buscar Alumno
5.Salir

Teclee su opcion (1-5): 3

ALUMNO No 1
--------------------------------
Direccion del elemento 0: 42A6D8
Carnet: c-01
Nombre: Juan
Edad: 22
Puntero al sig. borrado: 0
Estado del registro: o

ALUMNO No 2
--------------------------------
Direccion del elemento 1: 42A72C
Carnet: c-02
Nombre: Pedro
Edad: 21
Puntero al sig. borrado: 0
Estado del registro: o

ALUMNO No 3
--------------------------------
Direccion del elemento 2: 42A780
Carnet: c-03
Nombre: Maria
Edad: 18
Puntero al sig. borrado: 0
Estado del registro: o

ALUMNO No 4
--------------------------------
Direccion del elemento 3: 42A7D4
Carnet:
Nombre:
Edad: 0
Puntero al sig. borrado: 0
Estado del registro: b

cabecera_borrados apunta a: 42A7D4
Presione una tecla para continuar . . .
```

Ilustración 6-16: Estado de la lista después de insertar a un alumno. – Puntero Colgante

Como muestra la **Ilustración 6-16**, al insertar un nuevo elemento (al alumno Pedro), éste se inserta en la posición del primer elemento borrado, y se actualiza la cabecera de borrados. Así, la cabecera de borrados ahora pasa a apuntar al elemento 3 (alumno 4) del array, que está ubicado en la posición de memoria **42A7D4h**.

Opción 2: Borrar un alumno

Esta opción borra un alumno de la lista. Para borrar, se debe ingresar el nombre del estudiante que se desea borrar. Si el estudiante existe en la lista, entonces es borrado (se pone a valores vacíos), y se actualiza la cabecera de borrados para que ahora apunte a él mismo, de acuerdo a las siguientes condiciones:

1. El elemento que acaba de ser borrado, es el primero en ser borrado: cab_borrados que estaba apuntando a NULL, ahora pasa a apuntar a este elemento. La **Ilustración 6-17**, muestra lo expuesto.

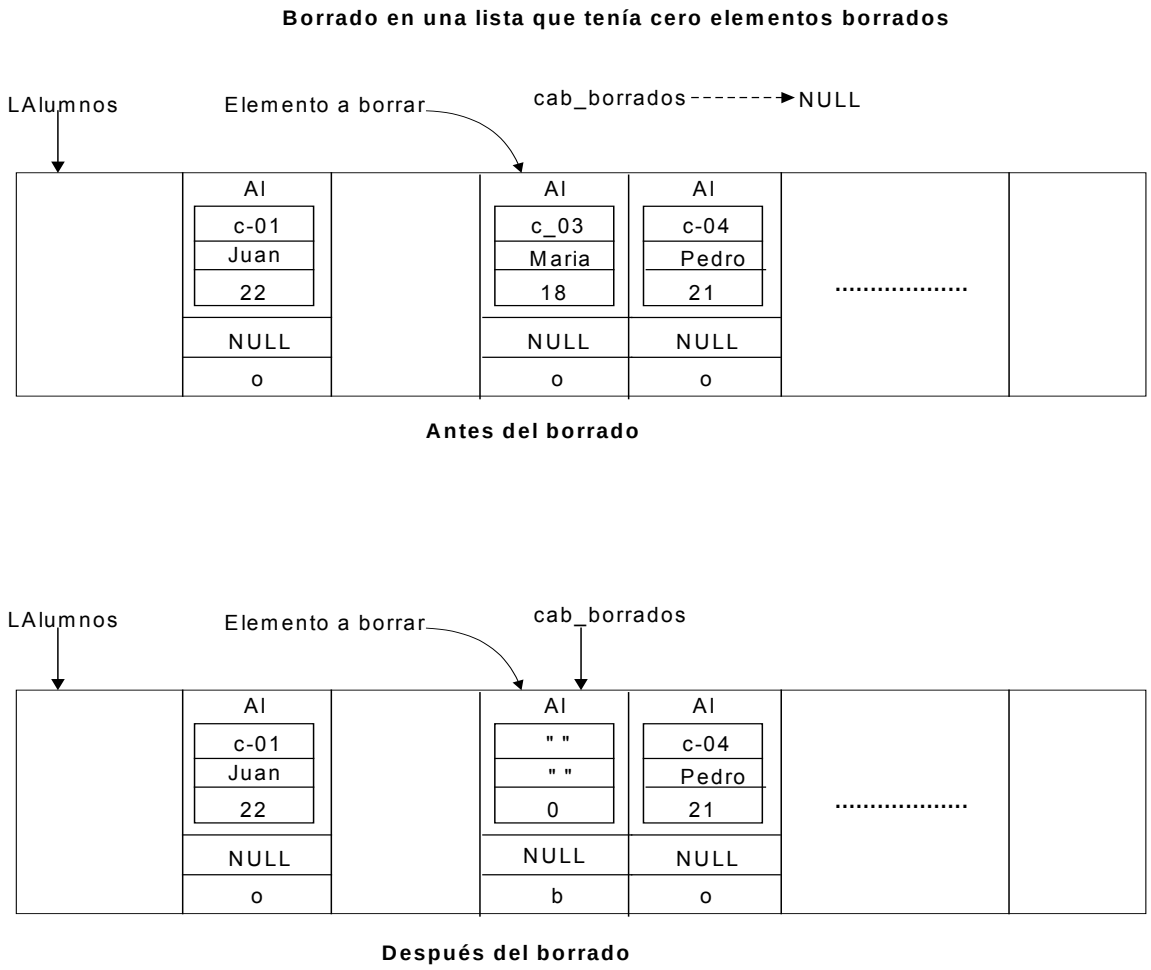
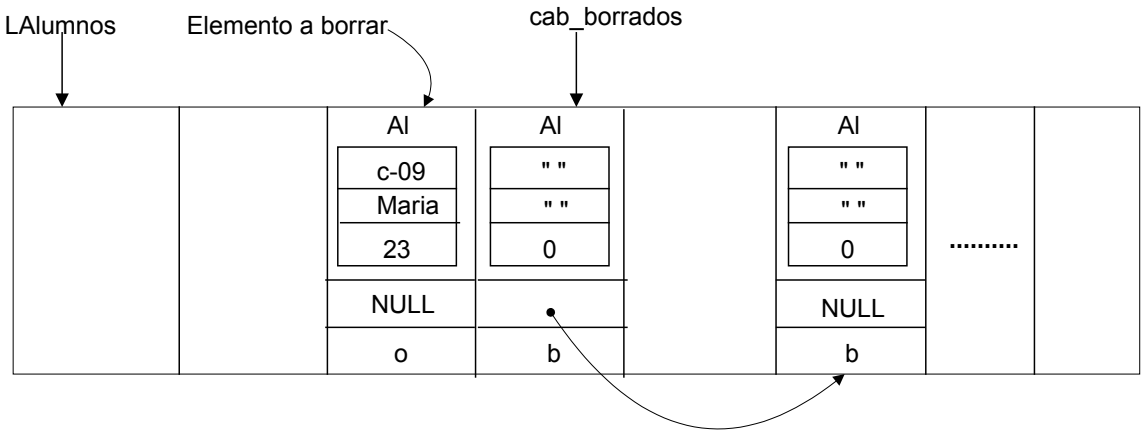


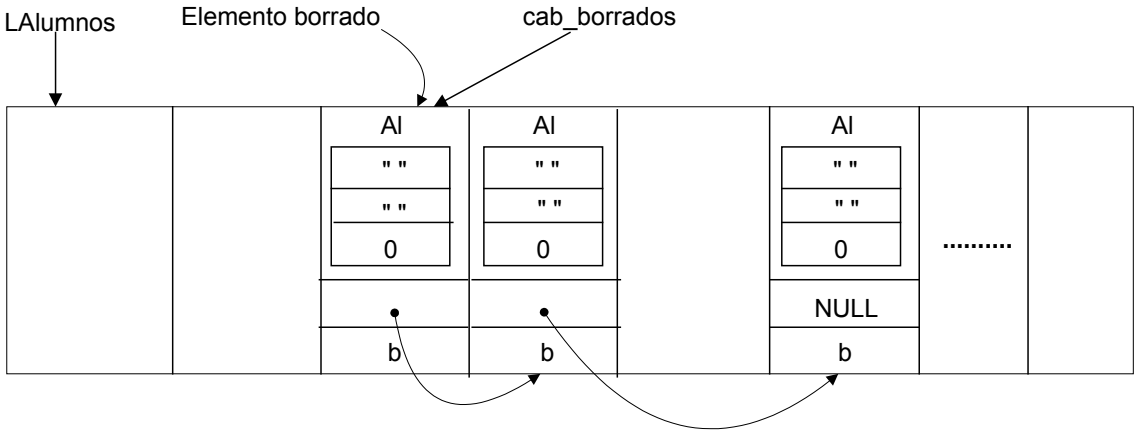
Ilustración 6-17: Borrado tipo 1. Puntero colgante

2. Ya existían elementos borrados y el que acaba de ser borrado, se encuentra **antes** del primer elemento borrado: **cab_borrado** apunta al elemento que acaba de ser borrado (que ahora es el primer borrado), y el puntero **sig_borrado** de dicho elemento ahora apunta al siguiente elemento borrado (que antes era el primero). La **Ilustración 6-18**, ilustra lo expuesto.

Borrado en una lista con al menos 1 elemento borrado: el elemento a borrar se encuentra antes del primer elemento borrado



Antes del borrado



Después del borrado

Ilustración 6-18: Borrado tipo 2. Puntero Colgante

3. Ya existían elementos borrados y el que acaba de ser borrado, está **después** del último elemento borrado: el puntero **sig_borrado** del último elemento que estaba borrado ahora pasa a apuntar al elemento que acaba de ser borrado (que ahora es el último borrado). La **Ilustración 6-19**, muestra lo expuesto.

Borrado en una lista con al menos 1 elemento borrado: el elemento a borrar se encuentra después del último elemento borrado

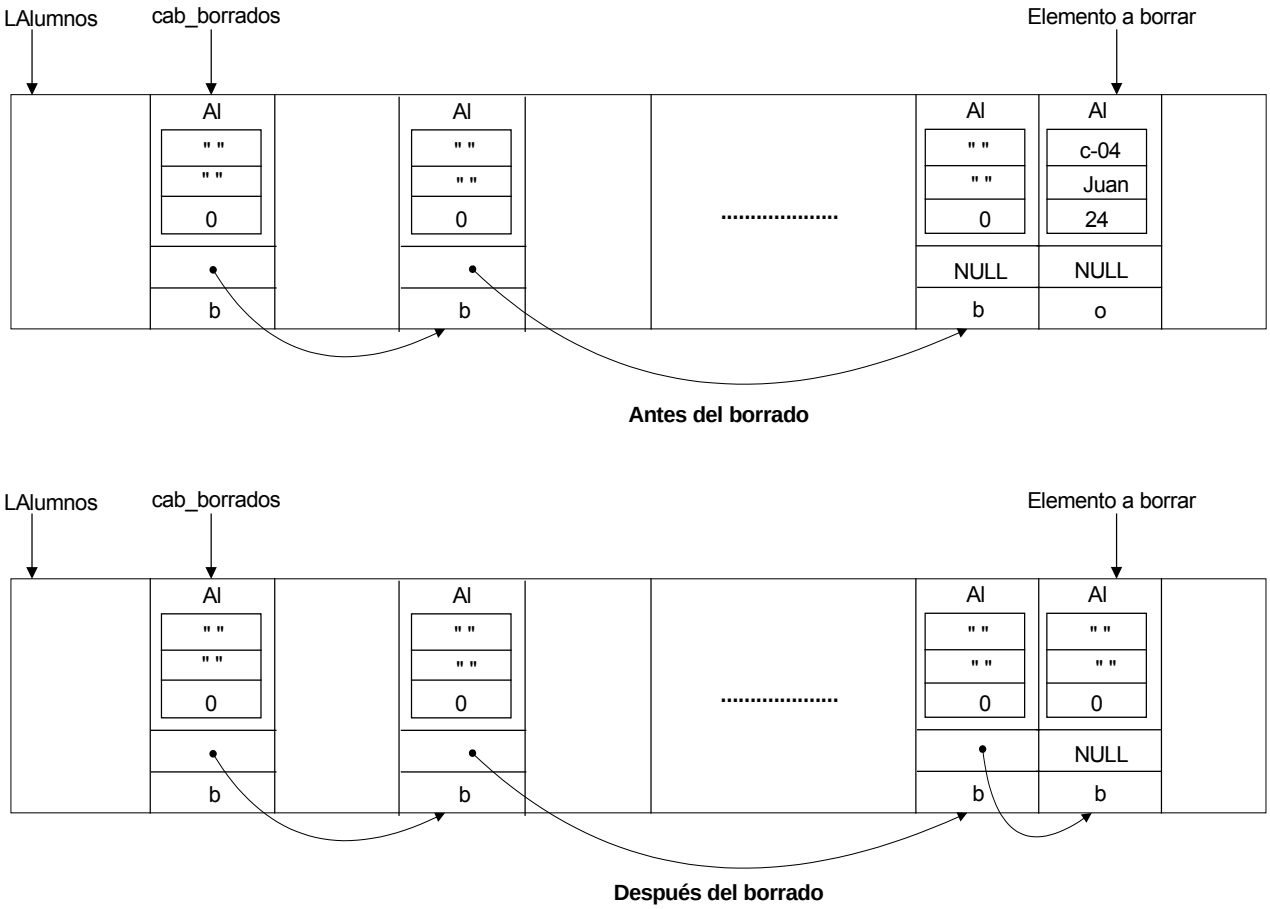


Ilustración 6-19: Borrado tipo 3. Puntero Colgante

4. Ya existían elementos borrados y el que acaba de ser borrado, está entre el primer y el último borrado: buscamos a la izquierda el primer elemento borrado y al encontrarlo, apuntamos desde éste al elemento que acabamos de borrar; luego buscamos a la derecha y al encontrarlo, apuntamos desde el elemento borrado a éste encontrado. La **Ilustración 6-20**, muestra lo expuesto.

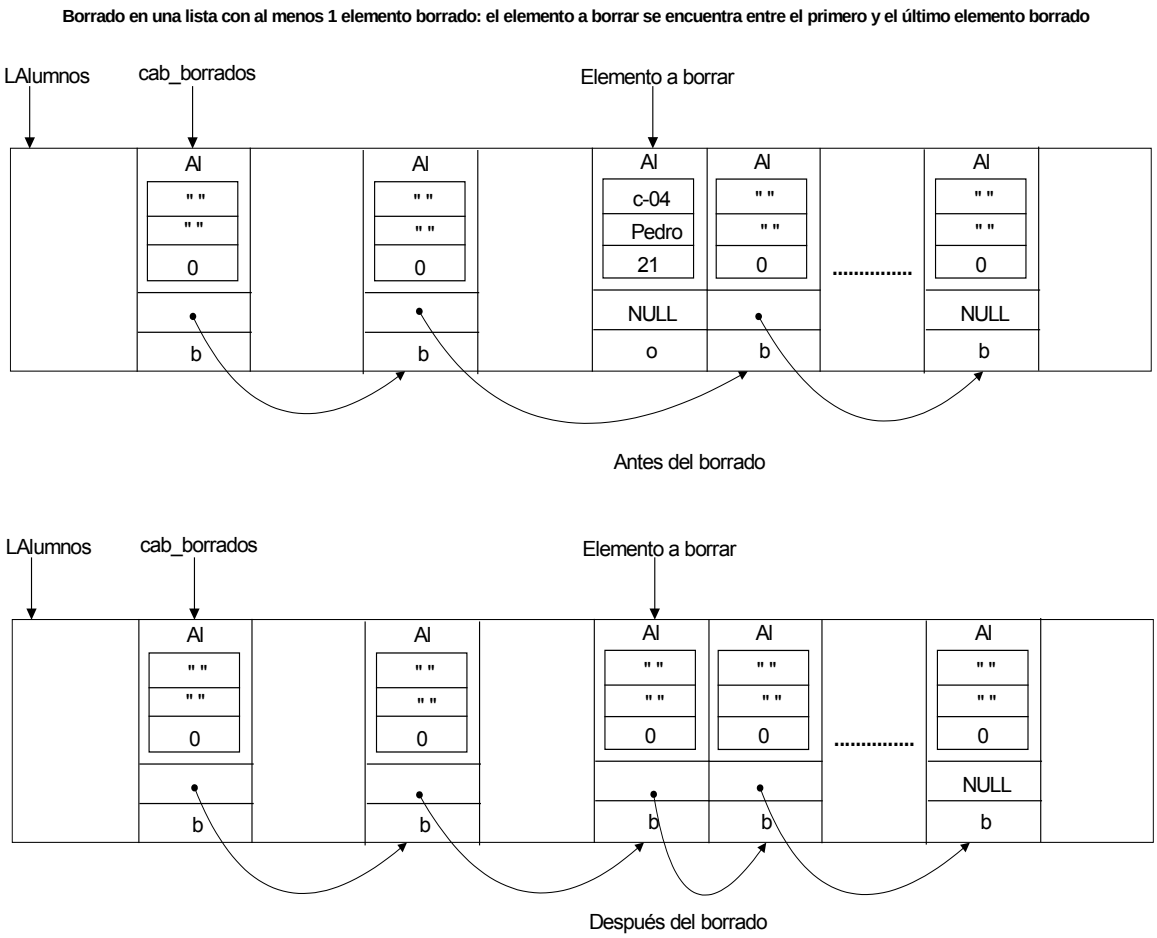


Ilustración 6-20: Borrado tipo 4. Puntero Cogante

En la **Ilustración 6-21**, se muestra el borrado del alumno Juan:

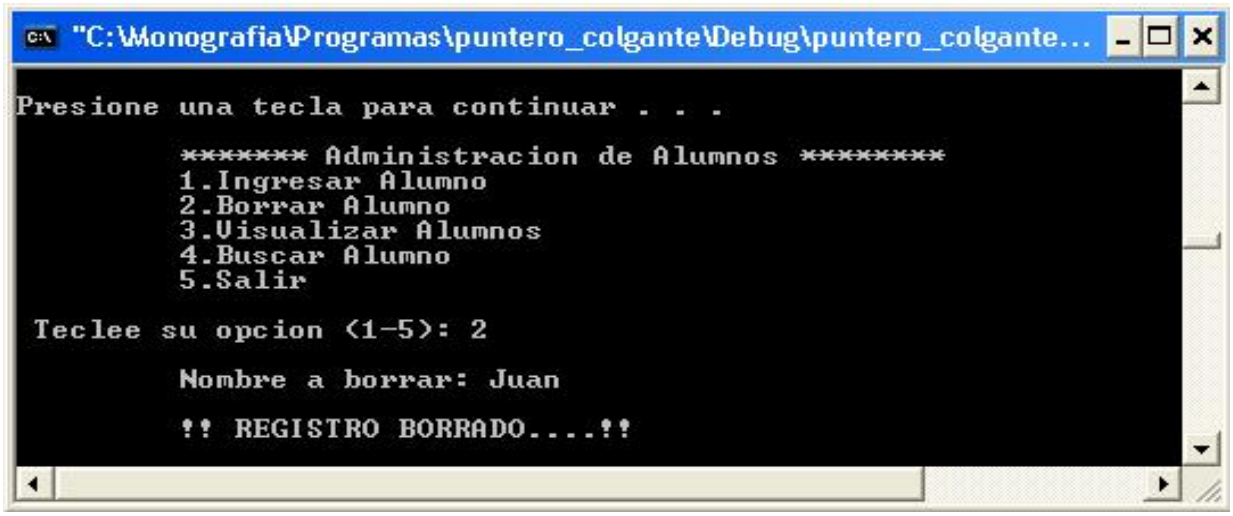


Ilustración 6-21: Borrado de un elemento. – Puntero Colgante

En la **Ilustración 6-22**, se muestra la lista después de eliminar a Juan.

```
******* Administracion de Alumnos ********
1.Ingresar Alumno
2.Borrar Alumno
3.Visualizar Alumnos
4.Buscar Alumno
5.Salir
Teclee su opcion (1-5): 3

ALUMNO No 1
--------------------------------
Direccion del elemento 0: 42A6D8
Carnet:
Nombre:
Edad: 0
Puntero al sig. borrado: 42A7D4
Estado del registro: b

ALUMNO No 2
--------------------------------
Direccion del elemento 1: 42A72C
Carnet: c-02
Nombre: Pedro
Edad: 21
Puntero al sig. borrado: 0
Estado del registro: o

ALUMNO No 3
--------------------------------
Direccion del elemento 2: 42A780
Carnet: c-03
Nombre: Maria
Edad: 18
Puntero al sig. borrado: 0
Estado del registro: o

ALUMNO No 4
--------------------------------
Direccion del elemento 3: 42A7D4
Carnet:
Nombre:
Edad: 0
Puntero al sig. borrado: 0
Estado del registro: b

cabecera_borrados apunt a: 42A6D8

Presione una tecla para continuar . . .
```

Ilustración 6-22: Listado de alumnos después de eliminar a Juan

La **opción 3** ha sido ya ejecutada antes.

La **opción 4**, busca un alumno en la lista. La **Ilustración 6-23**, muestra la búsqueda del alumno María.

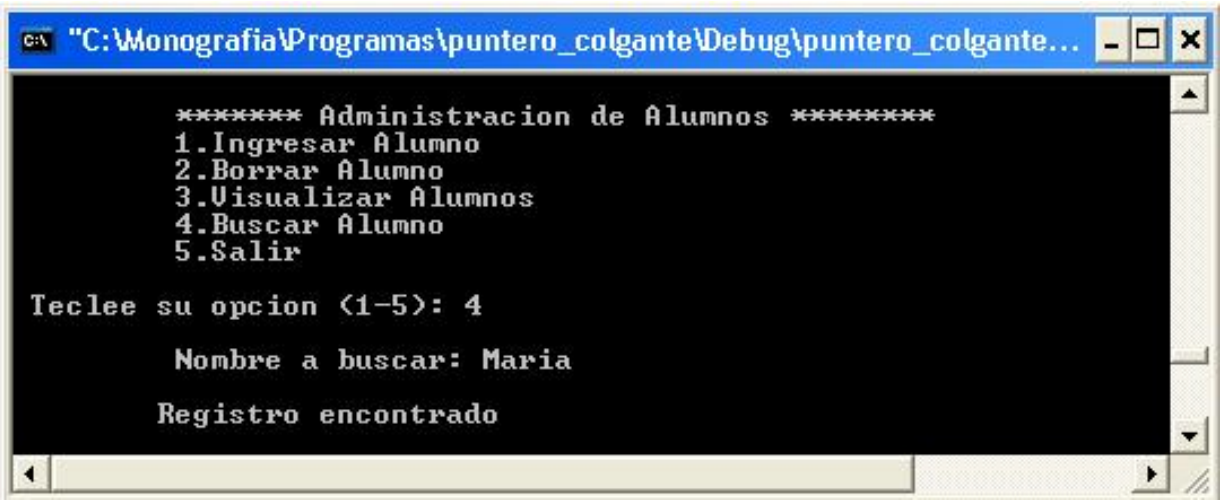


Ilustración 6-23: Buscar un alumno – Técnica del puntero colgante

La **opción 5**, guarda la lista en un archivo del disco duro y termina el programa.

A continuación mostraremos el código escrito en Lenguaje C, que implementa la técnica del puntero colgante.

/* Principal.c */

```
/* Implementación de la técnica de Organización de Archivos:
   Representación de archivos de longitud fija mediante el uso de punteros
   colgantes.
*/

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <crtdbg.h>
#include "definiciones.h"     // Declaraciones y definiciones de datos y funciones

// Función principal
void main()
{
        int opcion;
        Info_Alumno Alumno;
        char nombre_temp[50];
```

```
struct Lista aux;
int cont = 0;

// Incializar el array de almunos
InicializarAlumnos (LAlumnos);

pf = fopen("datos", "rb");  // abrir el fichero datos para leer
pf1 = fopen("inform", "rb");

if (pf != NULL)
{
        // El fichero existe, reconstruir la lista en MP
        // Traer datos del disco
        printf ("\n\t Trayendo datos del disco....");

        fread (&aux, tam_reg, 1, pf);
        while (!feof (pf))
        {
                if (ferror(pf))
                {
                        perror ("Error al leer del fichero\n");
                        exit (0);
                }

                LAlumnos[cont++] = aux;
                fread (&aux, tam_reg, 1, pf);
        }

        printf ("\n\t Datos traidos del disco....\n\n");

        fclose (pf);     // Cerramos el fichero

} // Fin de if (pf != NULL)

if (pf1 != NULL)
{
        i = getw(pf1);
        total_ingresados = getw(pf1);
        total_borrados = getw(pf1);

        if (ferror(pf1))
        {
                perror ("Error al leer del fichero\n");
                exit (0);
        }
        fclose (pf1);   // Cerramos el fichero

        ReconstruirLista (LAlumnos);
}
```

```
//i = cont;
while (1)
{
        opcion = Menu();

        switch (opcion)
        {
                fflush (stdin);
                case 1:         // Ingresar un estudiante al array
                        LeerDatosAlumno (&Alumno);
                        IngresarAlumno (LAlumnos, Alumno);
                        break;

                case 2:         // Borrar un estudiante del array
                        if (total_ingresados < 0 || i == 0)     // Verificar si hay
                                                                // alumnos en el array
                        {
                                printf ("\n\t No hay alumnos en la lista");
                                continue;
                        }

                        fflush (stdin);
                        printf ("\n\t Nombre a borrar: ");
                        gets (nombre_temp);
                        BorrarAlumno (LAlumnos, nombre_temp);
                        break;

                case 3:         // Visualizar alumnos
                        VisualizarAlumnos (LAlumnos);
                        break;

                case 4:         // Buscar a un estudiante en el array
                        if (total_ingresados < 0 || i == 0)     // Verificar si hay
                                                                // alumnos en el array
                        {
                                printf ("\n\t No hay alumnos en la lista");
                                continue;
                        }

                        fflush (stdin);
                        printf ("\n\t Nombre a buscar: ");
                        gets (nombre_temp);
                        BuscarAlumno (LAlumnos, nombre_temp);
                        break;


                case 5:         // Ingresar un estudiante al array
                        printf ("\n\t Iniciada la copia de datos al disco...");
```

```
                        GrabarEnDisco (LAlumnos);
                        printf ("\n\t Datos copiados al disco \n\n");
                        exit (0);
                        break;

                }

        }

} // Fin del main()
```

/* definciones.h */

```
////////////////////////////////////////////////////////////////////////////////
/////////////// Definiciones de estructuras de datos /////////////////
////////////////////////////////////////////////////////////////////////////////
#define ListaVacia (cabecera == NULL)

#define MAX_NUM_ELTOS 100    // Define el máximo número de estudiantes

typedef struct
{
        char carnet[20];        // Carnet del estudiante
        char nombre[50];        // Nombre del estudiante
        int edad;               // Edad del estudiante

}Info_Alumno;


struct Lista
{
        Info_Alumno Al;             // Información de un alumno
        struct Lista *sig_borrado;  // Puntero al siguiente registro eliminado
        char estado_reg;            // Estado del registro: 'b' --> borrado
                                    // 'o' --> ocupado
                                    // 'v' --> vacío
};

struct Lista LAlumnos[MAX_NUM_ELTOS];     // Array de alumnos
int i = 0;                                // llevar el conteo de la cantidad de
                                          // alumnos introducidos
struct Lista *cab_borrados = NULL;        // Apunta al primer elemento borrado
int total_borrados = 0;                   // total de registros borrados
int total_ingresados = 0;                 // total de registros ingreasados
FILE *pf;                                 // Puntero al fichero de datos
FILE *pf1;                                // Puntero al fichero de informacion
                                          // Contendrá, el valor de i,
                                          // total_borrados, total_ingresados

int tam_reg = sizeof (struct Lista);
```

```
////////////////////////////////////////////////////////////////////////////
/////////////////////////// Funciones Prototipos //////////////////////////
////////////////////////////////////////////////////////////////////////////

int  Menu();
void LeerDatosAlumno (Info_Alumno *pAl);
void IngresarAlumno (struct Lista *pListaAl, Info_Alumno Al);
void VisualizarAlumnos (struct Lista *pListaAl);
void InicializarAlumnos (struct Lista *pListaAl);
void BorrarAlumno (struct Lista *pListaAl, char *nombre);
void PonerValoresNulos (struct Lista *pL, int j);
void BuscarAlumno (struct Lista *pListaAl, char *nombre);
void GrabarEnDisco (struct Lista *pListaAl);
void ReconstruirLista (struct Lista *pListaAl);


////////////////////////////////////////////////////////////////////////////
/////////////////////////// Definiciones de Funciones //////////////////////
////////////////////////////////////////////////////////////////////////////
int Menu()
{
        int op;

        do
        {
                //system("cls");

                printf("\n\t ******* Administracion de Alumnos ********");
                printf("\n\t 1.Ingresar Alumno");
                printf("\n\t 2.Borrar Alumno");
                printf("\n\t 3.Visualizar Alumnos");
                printf("\n\t 4.Buscar Alumno");
                printf("\n\t 5.Salir");

                printf("\n\t\n Teclee su opcion (1-5): ");
                scanf("%d", &op);
        } while(op < 1 || op > 5);

        return (op);
}


// Leer datos de un alumno
void LeerDatosAlumno (Info_Alumno *pAl)
{
        fflush (stdin);
        printf ("\n\t **** DATOS DE ALUMNO ***");
        printf ("\n\t Carnet: ");
        gets (pAl->carnet);
```

```
        printf ("\n\t Nombre: ");
        gets (pAl->nombre);

        printf ("\n\t Edad: ");
        scanf ("%d", &pAl->edad);
}

// Ingresa un alumno a la lista
void IngresarAlumno (struct Lista *pListaAl, Info_Alumno Alumno)
{
        struct Lista *aux;
        // Verificar si no hay elementos en el array o, si hay elementos
        // en el array, pero ninguno se ha borrado
        if (cab_borrados == NULL || total_borrados == 0)
        {
                // Copiar el alumno Al, en la posición i de la lista
                pListaAl[i].Al = Alumno;
                pListaAl[i].sig_borrado = NULL;
                pListaAl[i].estado_reg = 'o'; // registro ocupado
                i++;
                total_ingresados++;
                return;
        }

        else    // Ya hay registros borrados
        {
                // La inserción del alumno, será en la posición del primer
                // elemento borrado; esta posición está dada por cab_borrados
                cab_borrados->Al = Alumno;
                cab_borrados->estado_reg = 'o';

                if (cab_borrados->sig_borrado == NULL)
                        // Había un sólo elemento borrado
                        cab_borrados = NULL;

                else
                {
                        aux = cab_borrados;
                        cab_borrados = cab_borrados->sig_borrado;
                        aux->sig_borrado = NULL;
                }


                total_ingresados++;

                if (total_borrados > 0)
                        total_borrados--;
```

```
                return;
        }
}

// Visualiza los alumnos que están en la lista
void VisualizarAlumnos (struct Lista *pListaAl)
{
        int j;

        printf ("\n");

        if (total_ingresados == 0)
        {
                printf ("\n\t !! NO HAY REGISTROS PARA VISUALIZAR !!");
                return;
        }

        for (j = 0; j < i; j++)
        {
                printf ("\n\t ALUMNO No %d", j + 1);
                printf("\n\t --------------------------------");
                printf ("\n\t Direccion del elemento %d: %X", j, &pListaAl[j]);
                printf ("\n\t Carnet: %s", pListaAl[j].Al.carnet);
                printf ("\n\t Nombre: %s", pListaAl[j].Al.nombre);
                printf ("\n\t Edad: %d", pListaAl[j].Al.edad);
                printf ("\n\t Puntero al sig. borrado: %X", pListaAl[j].sig_borrado);
                printf ("\n\t Estado del registro: %c", pListaAl[j].estado_reg);
                printf ("\n");

        }
        printf ("\n\t cabecera_borrados apunt a: %X \n\n", cab_borrados);

        system ("pause");
}

// Inicializar
void InicializarAlumnos (struct Lista *pListaAl)
{
        int j;

        for (j = 0; j < MAX_NUM_ELTOS; j++)
        {
                strcpy (pListaAl[j].Al.carnet, "");
                strcpy (pListaAl[j].Al.nombre, "");
                pListaAl[j].Al.edad = 0;

                pListaAl[j].sig_borrado = NULL;
                pListaAl[j].estado_reg = 'v'; // registro vacío
```

```
        }
}

////////////// Borra un alumno del array ////////////////////////
void BorrarAlumno (struct Lista *pListaAl, char *nombre)
{
        int j = 0, k;
        int primer_borrado, ultimo_borrado;
        bool reg_encontrado = false;
        struct Lista *aux;

        // Si el registro se encuentra, j contendrá la posición del elemento
        // a borrar
        for (j = 0; j < i; j++)
        {
                if ( strcmp (pListaAl[j].Al.nombre, nombre) == 0)
                {
                        reg_encontrado = true;
                        break;
                }
        }

        // Si el registro no se encontró...
        if (reg_encontrado == false)
        {
                printf ("\n\t !! REGISTRO NO ENCONTRADO !!");
                return;
        }


        // 1. Verificar que no hay registros borrados
        if (cab_borrados == NULL)  // también: if (total_borrados == 0)
        {
                cab_borrados = &pListaAl[j];        // cabecera apunta al primer reg
borrado
                PonerValoresNulos (pListaAl, j);
                printf ("\n\t REGISTRO BORRADO....\n");
                total_borrados++;     // incrementamos el total de borrados
                total_ingresados--;   // decrementamos el total de ingresados
                return;
        }

        if (cab_borrados != NULL)   // Ya hay registros borrados
        {
                // Habrá que considerar 3 casos:
                // 1. El registro a borrar está antes que el primer borrado
                // j posee el índice del valor a borrar

                // buscar la posición del primer elemento borrado
```

```
// posicion_borrado tiene la posición del primer elemento borrado
for (primer_borrado = 0; primer_borrado < i; primer_borrado++)
{
        if (pListaAl[primer_borrado].estado_reg == 'b')
                break;
}

// buscar la posición del último elemento borrado
// ultimo_borrado tiene la posición del primer elemento borrado
for (ultimo_borrado = i; ultimo_borrado >= 0; ultimo_borrado--)
{
        if (pListaAl[ultimo_borrado].estado_reg == 'b')
                break;
}


// Caso 1:
if (j < primer_borrado)         // El registro a borrar está antes del primer
                                // borrado
{
        PonerValoresNulos (pListaAl, j);
        printf ("\n\t !! REGISTRO BORRADO....!! \n");
        cab_borrados = &pListaAl[j];
        pListaAl[j].sig_borrado = &pListaAl[primer_borrado];
}

// Caso 2: El registro a borrar está después del último borrado
if (j > ultimo_borrado)
{
        pListaAl[ultimo_borrado].sig_borrado = &pListaAl[j];
        PonerValoresNulos (pListaAl, j);
        printf ("\n\t REGISTRO BORRADO....\n");
}

// Caso 3: El registro a borrar está entre el primero y el último borrado
if (j > primer_borrado && j < ultimo_borrado)
{
        // Buscar al elemento borrado más cercano a j, hacia atras
        // Dicho elemento será referenciado por k
        for (k = j - 1; k >= primer_borrado; k--)
        {
                if (pListaAl[k].estado_reg == 'b')
                        break;
        }

        // salvamos en aux, el valor de pListaAl[k].sig_borrado
        aux = pListaAl[k].sig_borrado;

        // Actualizar el sig borrado de k para que apunte a j
```

```
                    pListaAl[k].sig_borrado = &pListaAl[j];
                    PonerValoresNulos (pListaAl, j);

                    printf ("\n\t REGISTRO BORRADO....\n");

                    // j apunta a aux
                    pListaAl[j].sig_borrado = aux;

                } // fin de if (j > primer_borrado && j < ultimo_borrado)

                total_borrados++;     // aumentamos el total de borrados
                total_ingresados--;   // decrementamos el total de ingresados

                // Verificar si todos los registros introducidos han sido
                // borrados

        } // fin de if (cab_borrados != NULL)

        if (total_ingresados == 0)
                cab_borrados = NULL;

        return;

} // fin de BorrarAlumno
// Pone valores nulos en el elemento j del array
void PonerValoresNulos (struct Lista *pL, int j)
{
        strcpy (pL[j].Al.carnet, "");
        strcpy (pL[j].Al.nombre, "");
        pL[j].Al.edad = 0;
        pL[j].sig_borrado = NULL;
        pL[j].estado_reg = 'b';         // registo borrado

} // fin de PonerValoresNulos

// Buscar un alumno en la lista
void BuscarAlumno (struct Lista *pListaAl, char *nombre)
{
        bool encontrado = false;      // Nombre no encontrado
        int k;

        for (k = 0; k < i; k++)
        {
                if (strcmp(pListaAl[k].Al.nombre, nombre) == 0)
                {
                        encontrado = true;
                        break;
                }
```

```
        }

        if (encontrado == false)
        {
                printf("\n\tRegistro no encontrado\n");
                return;
                system("pause");
        }

        printf("\n\tRegistro encontrado\n");
        return;
        system("pause");

}

// Graba la lista al archivo datos en el disco duro
void GrabarEnDisco (struct Lista *pListaAl)
{
        int k;

        pf = fopen("datos", "wb");    // abrir el fichero para escribir

        if (pf == NULL)
        {
                perror("Error al crear el archivo");
                exit (0);
        }

        // Verificar si la lista está vacía
        if (total_ingresados == 0 || i == 0)
        {
                printf ("\n\t LISTA VACIA!!!");
                return;
        }

        for (k = 0; k < i; k++)
                fwrite (&pListaAl[k], tam_reg, 1, pf);

        pf1 = fopen("inform", "wb"); // abrir el fichero para escribir

        if (pf1 == NULL)
        {
                perror("Error al crear el archivo");
                exit (0);
        }

        // Escribir en pf1 el valor de i, total_ingresados y total_borrados
        putw (i, pf1);
        putw (total_ingresados, pf1);
```

```
        putw (total_borrados, pf1);

        // Cerrar el archivo

        fclose (pf);
        fclose (pf1);

}

// Reconstruir la lista (con sus punteros a elementos borrados)
void ReconstruirLista (struct Lista *pListaAl)
{
        int j, k;
        bool borrado_encontrado = false;
        struct Lista *aux;

        for (j = 0; j < i; j++)
        {
                if (pListaAl[j].estado_reg == 'b')
                {
                        borrado_encontrado = true;
                        break;
                }
        }

        if (borrado_encontrado == false)    // No hay elementos borrados
        {
                cab_borrados = NULL;
                return;
        }


        // Si llegamos aquí, es que hay al menos un elemento borrado
        // Apuntar a ese elemento borrado, con cab_borrados
        cab_borrados = &pListaAl[j];

        aux = cab_borrados;

        for (k = j; k < i; k++)
        {
                if (pListaAl[k].estado_reg == 'b')
                {
                        aux->sig_borrado = &pListaAl[k];
                        aux = aux->sig_borrado;
                }
        }       // fin del for
}       // fin de la función
```

Algoritmos de las principales funciones de la técnica del puntero colgante
Algoritmo de la función IngresarAlumno

```
Function IngresarAlumno (struct Lista *pListaAl, Info_Alumno Alumno) void

Var aux: pointer to Lista
[Verificar si no hay elementos en el array o, si hay elementos en el array, pero ninguno se ha borrado]

1. Si cab_borrados = NULL o total_borrados = 0, entonces
        [Copiar el alumno Al, en la posición i de la lista]
        pListaAl[i].Al ← Alumno
        pListaAl[i].sig_borrado ← NULL
        pListaAl[i].estado_reg ← 'o';          [registro ocupado]
        Incrementar i
        Incrementar total_ingresados
        Salir de la función
FinSi

2. SiNo         [Ya hay registros borrados]
        [La inserción del alumno, será en la posición del primer elemento borrado; esta posición está dada por cab_borrados]
        cab_borrados->Al ← Alumno
        cab_borrados->estado_reg ← 'o'
        Si cab_borrados->sig_borrado = NULL
                [Había un sólo elemento borrado]
                cab_borrados ← NULL
        FinSi
        SiNo
                aux ← cab_borrados
                cab_borrados ← cab_borrados->sig_borrado
                aux->sig_borrado ← NULL
        FinSiNo
        total_ingresados ← total_ingresados + 1
        Si total_borrados > 0
                total_borrados ← total_borrados - 1
        FinSi
        Salir de la función
FinSiNo
FinFunction
```

Algoritmo de la función BorrarAlumno

```
Function BorrarAlumno (struct Lista *pListaAl, char *nombre) void
Var j, k, primer_borrado, ultimo_borrado integer
Var reg_encontrado ← false
Var aux pointer to Lista

[Si el registro se encuentra, j contendrá la posición del elemento a borrar]
1. Para j ← 0, j < i, j++
```

```
        Si pListaAl[j].Al.nombre = nombre, entonces
                reg_encontrado ← true
                Salir del for
        FinSi
FinPara

[Si el registro no se encontró]
2. Si reg_encontrado = false, entonces
        Imprimir "REGISTRO NO ENCONTRADO"
        Salir de la función
FinSi

[Verificar que no hay registros borrados]
3 Si cab_borrados = NULL
        cab_borrados ← &pListaAl[j]        [cabecera apunta al primer reg borrado]
        PonerValoresNulos (pListaAl, j)
        Imprimir "REGISTRO BORRADO"
        total_borrados ← total_borrados + 1        [incrementamos el total de borrados]
        total_ingresados ← total_ingresados – 1  [decrementamos el total de
ingresados]
        Salir de la función
FinSi

4. Si cab_borrados <> NULL        [Ya hay registros borrados]
[Buscar la posición del primer elemento borrado]
[posicion_borrado tiene la posición del primer elemento borrado]
        Para primer_borrado ← 0, primer_borrado < i, primer_borrado++
                Si pListaAl[primer_borrado].estado_reg = 'b'
                        Regresar de la función
        FinPara
        [Buscar la posición del último elemento borrado]
        [ultimo_borrado tiene la posición del primer elemento borrado]
        Para ultimo_borrado ← i, ultimo_borrado >= 0, ultimo_borrado--
                Si pListaAl[ultimo_borrado].estado_reg = 'b'
                        Regresar de la función
        FinPara
        [Caso 1]
        [El registro a borrar está antes del primer borrado]
        Si j < primer_borrado
                PonerValoresNulos (pListaAl, j)
                Imprimir !! REGISTRO BORRADO....!!
                cab_borrados ← &pListaAl[j]
                pListaAl[j].sig_borrado ← &pListaAl[primer_borrado]
        FinSi
        [Caso 2: El registro a borrar está después del último borrado]
        Si j > ultimo_borrado
                pListaAl[ultimo_borrado].sig_borrado ← &pListaAl[j]
                PonerValoresNulos (pListaAl, j)
                Imprimir !! REGISTRO BORRADO.!!
```

```
        FinSi
        [Caso 3: El registro a borrar está entre el primero y el último borrado]
        Si j > primer_borrado && j < ultimo_borrado
                [Buscar al elemento borrado más cercano a j, hacia atras
                Dicho elemento será referenciado por k]
                Para k = j – 1, k >= primer_borrado, k—
                        Si pListaAl[k].estado_reg = 'b'
                                Salir
                        FinSi
                FinPara
                [Salvamos en aux, el valor de pListaAl[k].sig_borrado]
                aux ← pListaAl[k].sig_borrado
                [Actualizar el sig borrado de k para que apunte a j]
                pListaAl[k].sig_borrado ← &pListaAl[j]
                PonerValoresNulos (pListaAl, j)
                Imprimir !! REGISTRO BORRADO.!!
                [j apunta a aux]
                pListaAl[j].sig_borrado ← aux
        FinSi j > primer_borrado && j < ultimo_borrado
        Incrementar total_borrados [Aumentamos el total de borrados]
        Decrementar total_ingresados      [Decrementamos el total de ingresados]
        [Verificar si todos los registros introducidos han sido borrados]
FinSi cab_borrados <> NULL
Si total_ingresados = 0
        cab_borrados ← NULL
Regresar a la función
Fin de la Function BorrarAlumno
```

6.2 Registros de longitud variable

Los registros de longitud variables surgen de varias maneras en los sistemas de Base de Datos; estos se distinguen por:

- Almacenamiento de varios tipos de registros en un mismo archivo.
- Tipos de registros que permiten longitudes variables para uno o varios de los campos.
- Tipos de registros que permiten campos repetidos.

Ejemplo: para mostrar las diversas técnicas de implementación, consideremos la siguiente definición de registro:

```
Type lista_cuentas = record
Nombre_suc : char (22);
Información_cuenta : array[1...α] of record;
        Numero_cuenta : char [10];
        Saldo : real;
        End
End
```

Se define información_cuenta como un array con un número arbitrario de elementos, por lo que no hay ningún limite para el tamaño que pueden tener los registros (Hasta el tamaño del disco, ¡por supuesto !).

Los registros de longitud variable pueden ser representados como: cadenas de bytes y como longitud fija.

6.2.1 Representación en cadena de bytes

Un método sencillo es adjuntar un símbolo de fin de registro (⊥) al final de cada registro. Así, se puede guardar cada registro como una cadena de bytes consecutivos. La **Tabla** 6-5, muestra un ejemplo de esta técnica.

Tabla 6-5: Representación en cadena de bytes

Registro	N_suc	Nº_cuenta	Saldo					
0	Mianus	C-102	400	C-201	900	C-218	700	⊥
1	Becerril	C-305	500	⊥				
2	Centro	C-215	650	⊥				
3	Brighton	C-121	700	C-110	600	⊥		
4	Avalcard	C-222	800	⊥				
5	BDF	C-350	950	⊥				

Inconvenientes

- No resulta sencillo volver a utilizar el espacio ocupado anteriormente por un registro borrado. Generan gran número de fragmentos pequeños de almacenamientos en disco desaprovechado.

- Problema de crecimiento, por lo general no queda espacio para el aumento de los tamaños de los registros.

Codificación en Lenguaje C de la técnica Representación en Cadena de Bytes

A continuación, veremos una propuesta de programa en Lenguaje C, que implementa la técnica anteriormente descrita.

Usaremos las siguientes estructuras para manipular los registros:

```
typedef struct
{
        char  No_cta[20];     // Número de cuenta
        float  saldo;         // Saldo de la cuenta
        char  estado_reg;     // 'b' ->Borrado;'o' -> Lleno

}Info_Cuenta;


struct Cuenta
{
        char  nom_suc [100];        // Nombre de la sucursal
        struct  Cuenta *p_sig;      // Puntero al siguiente Elemento
        int  cant_ctas;             // cantidad de cuentas
        Info_Cuenta  *pIC;          // Puntero a Info_Cuenta
};
```

La explicación de cada uno de los miembros de las estructuras anteriores, se detallan a continuación:

Estructura Info_Cuenta:

Campo	Descripción
No_cta	Especifica el número de cuenta que maneja cada sucursal.
saldo:	Especifica el saldo de la cuenta representada por No_cta.
estado_reg	Proporciona información sobre el estado de la cuenta.

Estructura Cuenta:

Campo	Descripción
nom_suc	Representa al nombre de la sucursal.
p_sig	Es un puntero a una estructura del mismo tipo, que apuntará al siguiente elemento.
cant_cuentas	Representa la cantidad de cuentas que tiene cada sucursal.
pIC	Es un puntero a Info_Cuenta. Apunta a un array de cuentas de la sucursal.

La idea será llevar una lista lineal enlazada; cada elemento de la lista será un elemento de tipo **struct Cuenta**. Por cada elemento de la lista, el campo miembro apuntador **pIC** apuntará a un array de todas las cuentas de dicha sucursal.

En la **Ilustración 6-24** se puede apreciar un ejemplo de instanciación de lo expuesto anteriormente:

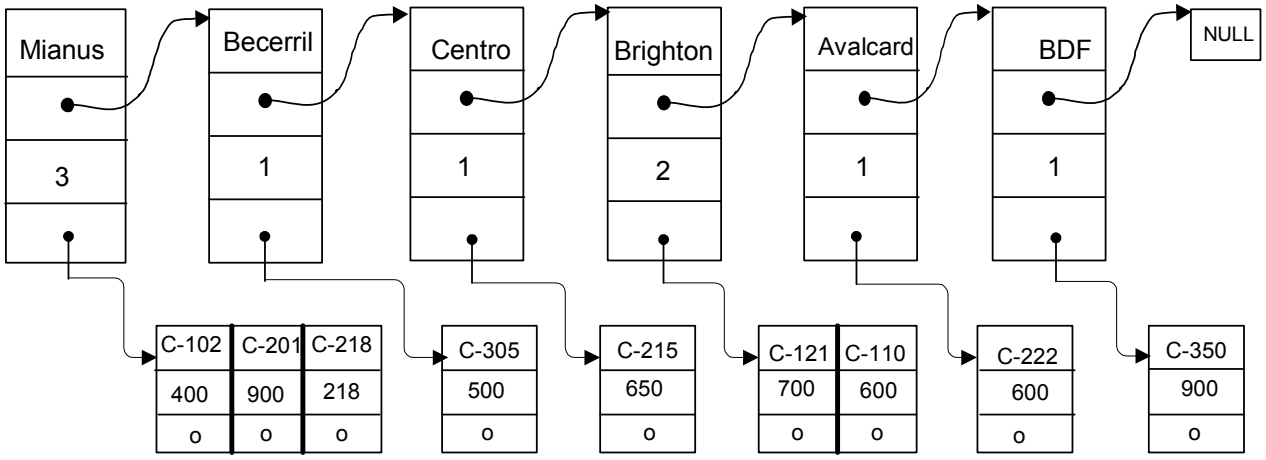


Ilustración 6-24: Lista lineal ejemplificando la representación en cadena de bytes

En la **Ilustración 6-24**, se puede ver como cada sucursal posee una cantidad de cuentas variables, por lo que, ***los tamaños en bytes serán distintos para cada elemento que represente a una sucursal;*** esto es en esencia, la idea de la representación de los archivos de longitud variable mediante la técnica de cadena de bytes.

La ejecución del programa mostrado en la **Ilustración 6-25**, presenta un menú de posibles operaciones con las sucursales de distintos bancos:

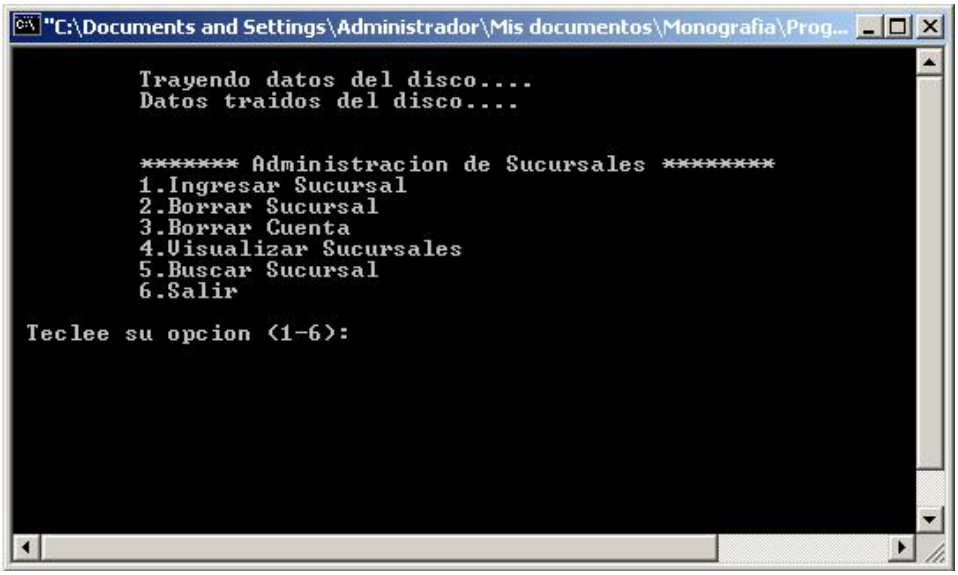


Ilustración 6-25: Menú de Sucursales

La explicación de cada una de las opciones del menú, se detallan a continuación:

Opción 1: Al teclear la opción 1, el programa pedirá al usuario información sobre la nueva sucursal a ingresar, la cual será ingresada a la lista lineal después del último elemento introducido (al principio si la lista está vacía). Esta forma de inserción es similar al funcionamiento de una cola; los registros se introducen uno a uno después del último introducido. La **Ilustración 6-26** muestra lo anteriormente expuesto.

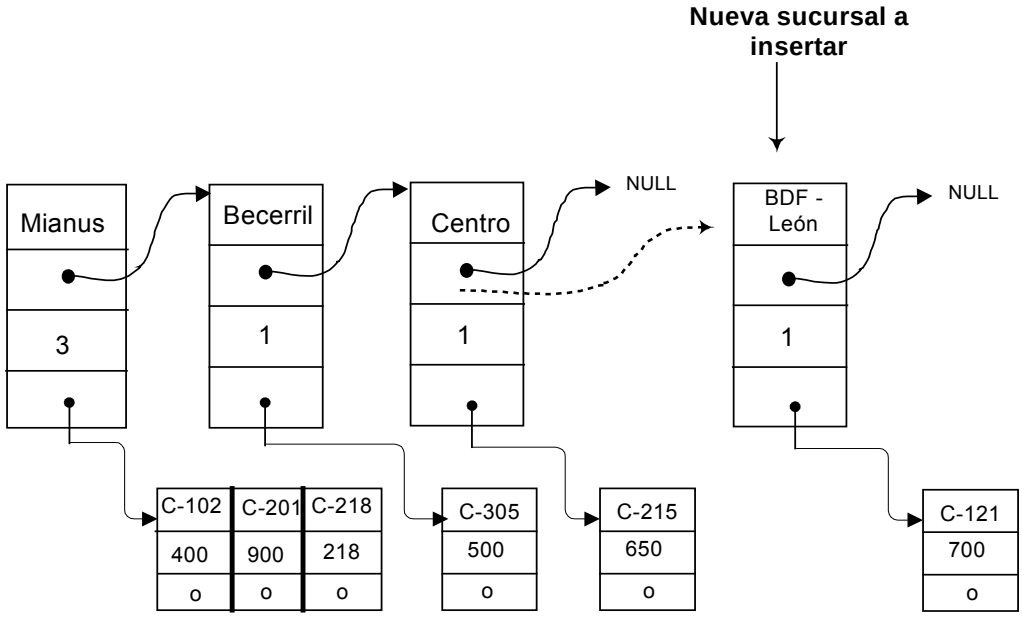


Ilustración 6-26: Inserción de una nueva sucursal

La interfaz presentada al usuario cuando desde el menú teclea la opción 1, se muestra en la **Ilustración 6-27**.

Ilustración 6-27: Ingreso de una nueva sucursal

A como se puede apreciar en la **Ilustración 6-27**, luego de pedir el nombre de la sucursal, se pregunta por la cantidad de cuentas que maneja esta sucursal. De acuerdo a la cantidad de cuentas tecleadas, por cada cuenta, se preguntará el número de cuenta y el saldo. Una vez más se puede observar que, cada sucursal manejará cantidades de cuentas variables, de ahí el tamaño variable de cada elemento estructura que representa a una sucursal.

Opción 2: Esta opción borra una sucursal y las cuentas pertenecientes a la misma. El elemento dentro de la lista lineal que representa a la sucursal, es eliminado y los punteros se reasignan. A manera de ejemplo, la **Ilustración 6-28** muestra las sucursales visualizadas, para después borrar la sucursal BAC (**Ilustración 6-29**) y después ser visualizadas de nuevo (**Ilustración 6-30**).

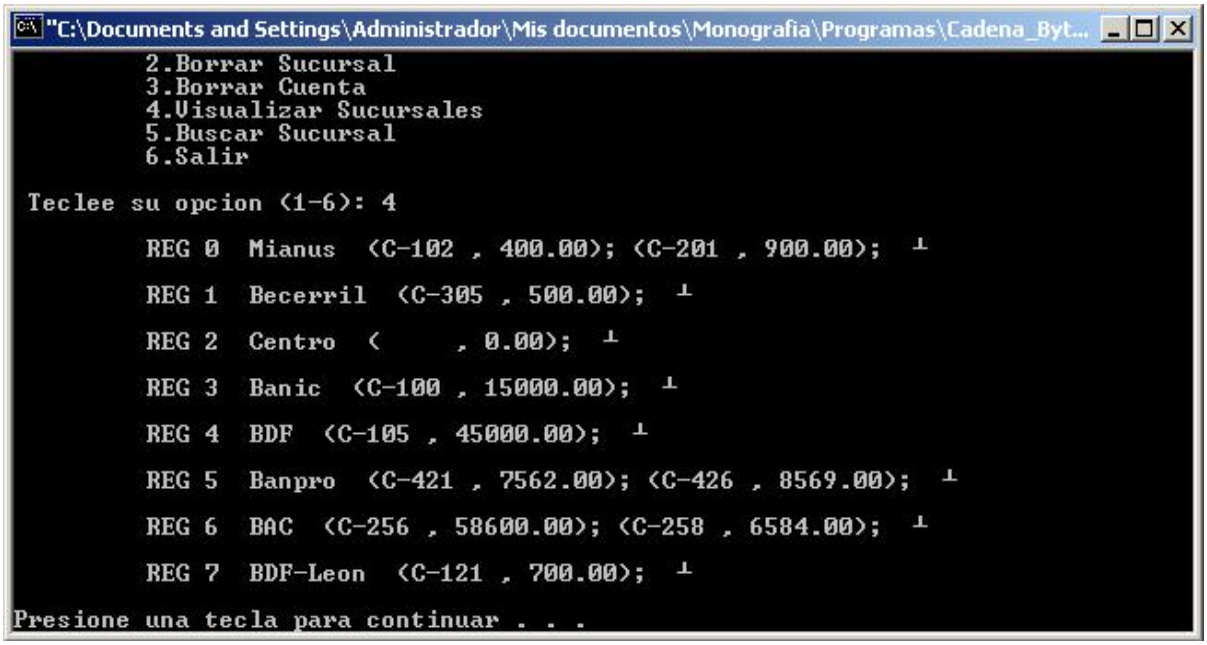


Ilustración 6-28: Sucursales listadas

La **Ilustración 6-29** muestra el borrado de la sucursal BAC.

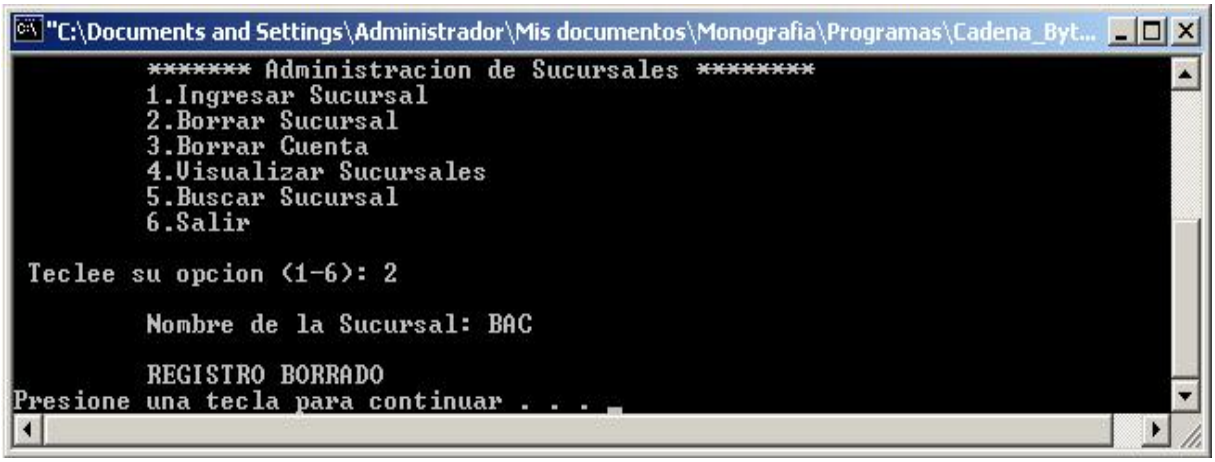


Ilustración 6-29: Mensaje de borrado de la sucursal BAC

La **Ilustración 6-30** muestra el listado de las sucursales después de borrar a la sucursal BAC.

Un aspecto importante a señalar es que, la visualización de cada sucursal tiene el siguiente formato:

Número de Registro Nombre de la Sucursal (Num_Cta1, Saldo 1); (Num_Cta2, Saldo 2).....

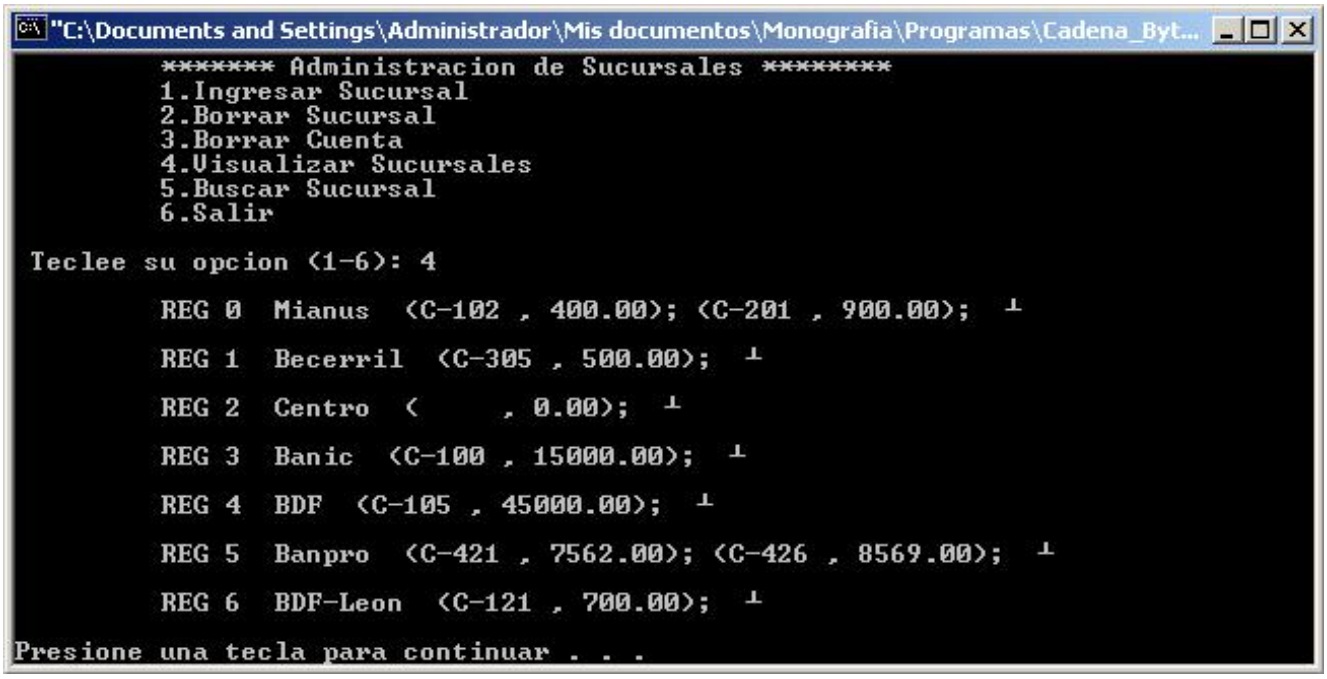


Ilustración 6-30: Lista de sucursales después de borrar a la sucursal BAC

Opción 3: La opción 3 borra una cuenta de una sucursal. La **Ilustración 6-31**, muestra cómo se borra la **cuenta C-421** de la sucursal **Banpro**.

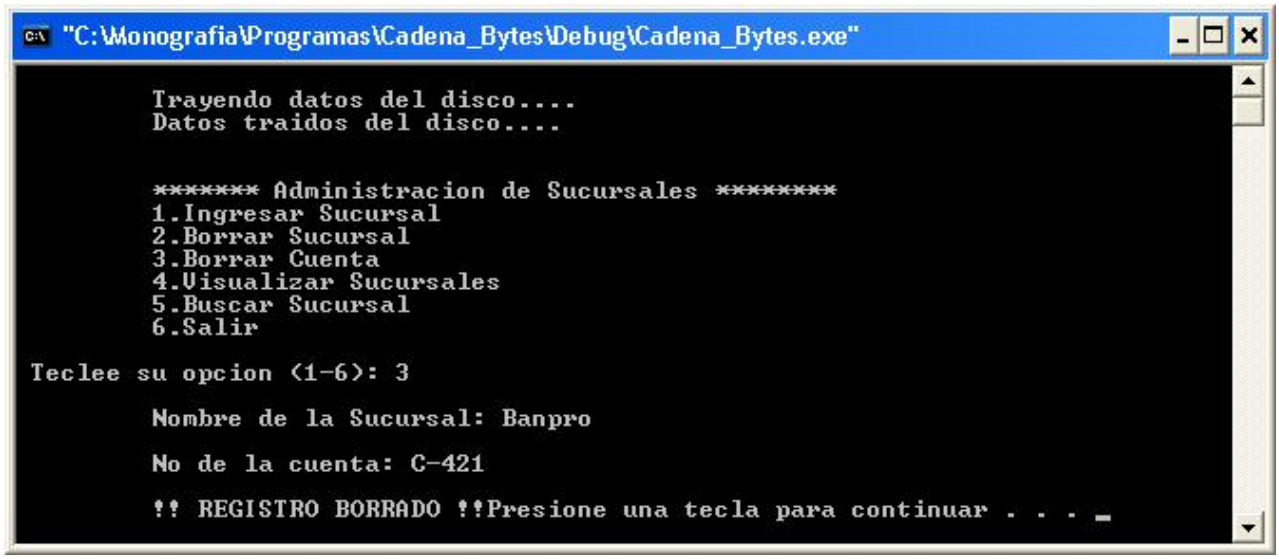


Ilustración 6-31: Borrando la cuenta C-421 de la sucursal Banpro

Cabe destacar que, al borrar una cuenta, se localiza dicha cuenta en la sucursal dada, y al encontrarla, se ponen valores vacíos en sus campos número de cuenta y saldo.

Opción 4: La opción 4 es visualizar sucursales y ya fue ejecutada anteriormente.

Opción 5: Busca una sucursal. Una ejecución de esta orden del menú, se muestra en la **Ilustración 6-32** (visualiza las sucursales) y en la **Ilustración 6-33**, busca una sucursal:

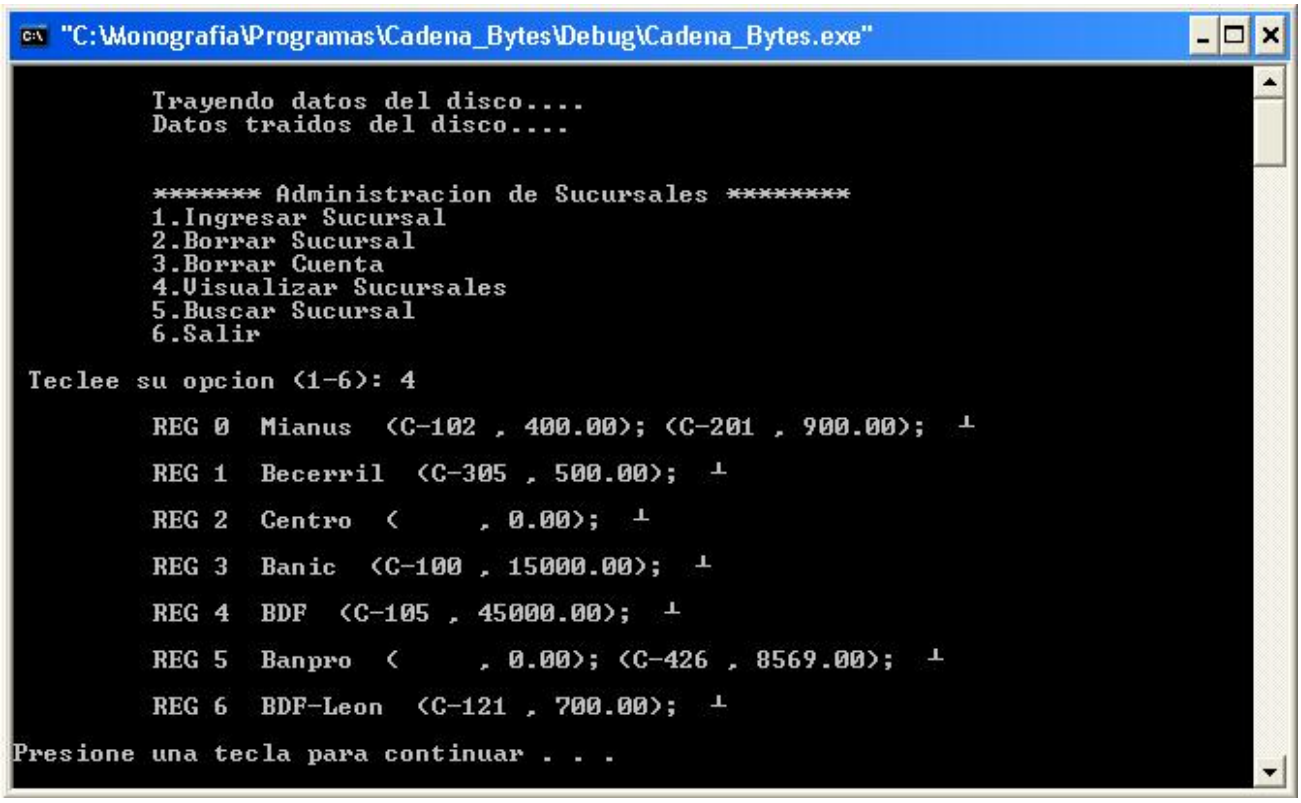


Ilustración 6-32: Visualización de sucursales

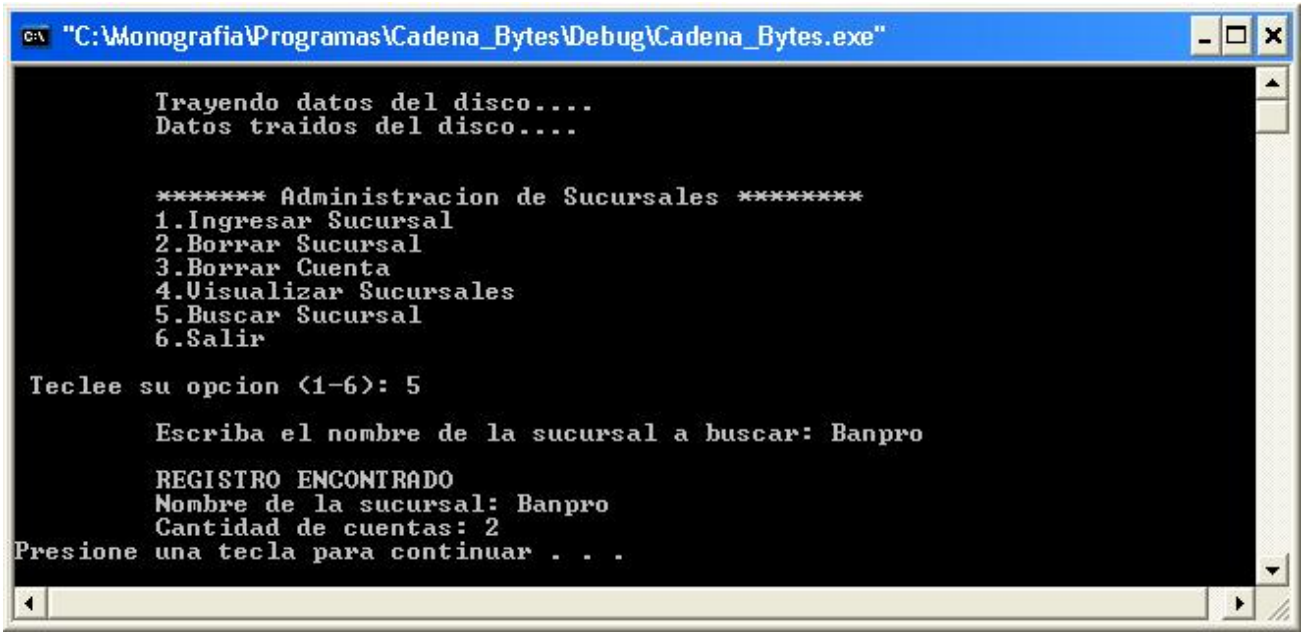


Ilustración 6-33: Búsqueda de una cuenta dado el nombre de una sucursal

A continuación, mostramos el código en Lenguaje C, que ilustra la técnica de representación en cadena de bytes para archivos con registros de longitud variable.

/* Programa que implementa la técnica de representación en cadena de bytes, para archivos con registros de longitud variable. */
/* Principal.c */

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <crtdbg.h>
#include "definiciones.h"

// Función principal
void main()
{
        int opcion;
        int cant_leidos;
        struct Cuenta *p = NULL, *f = NULL;        // Principio y final de la cola
        char nom_suc[100];
        char No_cuenta[20];
        struct Cuenta aux;


        pf = fopen("datos", "rb");  // abrir el fichero datos para leer
```

```
if (pf != NULL)
{
        // El fichero existe, reconstruir la lista en MP
        // Traer datos del disco
        printf ("\n\t Trayendo datos del disco....");

        //rewind (pf);
        cant_leidos = fread (&aux, tam_reg, 1, pf);
        //bytes_totales += tam_reg;

        while (!feof (pf))
        {
                //fseek (pf, (long)bytes_totales, SEEK_SET);
                if (ferror(pf))
                {
                        perror ("Error al leer del fichero\n");
                        exit (0);
                }

                IntroducirListaNueva (&p, &f, aux);
                cant_leidos = fread (&aux, tam_reg, 1, pf);
                bytes_totales += tam_reg;

        }

        if (ferror(pf))
        {
                perror ("Error al leer del fichero");
                exit (0);
        }


        printf ("\n\t Datos traidos del disco....\n\n");

        fclose (pf);    // Cerramos el fichero

}

while (1)
{
        opcion = Menu();

        switch (opcion)
        {
                case 1:                 // Ingresar nueva sucursal
                        fflush(stdin);
                        printf("\n\t Nombre de la sucursal: ");
                        gets(nom_suc);
                        IngresarNuevaSucursal (&p, &f, nom_suc);
```

```
                    break;

                case 2:         // Borrar sucursal
                    fflush(stdin);
                    printf("\n\t Nombre de la Sucursal: ");
                    gets(nom_suc);
                    BorrarSucursal(&p, &f, nom_suc);
                    break;

                case 3:         // Borrar Cuenta
                    fflush(stdin);
                    printf("\n\t Nombre de la Sucursal: ");
                    gets(nom_suc);

                    printf("\n\t No de la cuenta: ");
                    gets(No_cuenta);
                    BorrarCuenta (&p, nom_suc, No_cuenta);
                    break;

                case 4:
                    Visualizar(p);
                    system("pause");
                    printf("\n");
                    break;

                case 5:
                    fflush (stdin);
                    printf ("\n\t Escriba el nombre de la sucursal a buscar: ");
                    gets (nom_suc);
                    BuscarSucursal(p, nom_suc);
                    system("pause");
                    printf("\n");
                    break;


                case 6:
                    printf ("\n\t Iniciada la copia de datos al disco...");
                    GrabarEnDisco (p);
                    printf ("\n\t Datos copiados al disco \n\n");

                    exit(0);
                    break;

        } // fin switch

} //fin del while

EliminarLagunas(&p);
```

```
        if(_CrtDumpMemoryLeaks())
                printf("Hay lagunas de memoria");

} // fin del main
```

```
/* defincciones.h */
/* Declaraciones de datos y funciones, y definiciones de funciones */

#define ListaVacia (cabecera == NULL)

typedef struct
{
        char No_cta[20];      // Número de cuenta
        float saldo;          // Saldo de la cuenta
        char estado_reg;      // 'b' ->Borrado;'o' -> Lleno

}Info_Cuenta;

struct Cuenta
{
        char nom_suc [100];         // Nombre de la sucursal
        struct Cuenta *p_sig;       // Puntero al siguiente Elemento
        int cant_ctas;              // cantidad de cuentas
        Info_Cuenta *pIC;           // Puntero a Info_Cuenta
};

int tam_reg = sizeof (struct Cuenta);   // tamaño de la estructura
int tam_InfoCuenta = sizeof (Info_Cuenta);
int bytes_totales = 0;

////////////////////////////////////// Funciones Prototipos ////////////////////////////////////////////////
int Menu();
void IngresarNuevaSucursal(struct Cuenta **p, struct Cuenta **f, char *nom_suc);
struct Cuenta *NuevoElemento();
void Error (void);
void Visualizar (struct Cuenta *p);
void EliminarLagunas (struct Cuenta **pc);
void BorrarCuenta (struct Cuenta **p, char *nom_suc, char *No_cuenta);
void BorrarSucursal (struct Cuenta **p, struct Cuenta **f, char *nom_suc);
void GrabarEnDisco (struct Cuenta *cab);
void IntroducirListaNueva (struct Cuenta **p, struct Cuenta**f, struct Cuenta aux);
void BuscarSucursal (struct Cuenta *cab, char *ns);

FILE *pf;

////////////////////////////////////////////// Definición de Funciones ////////////////////////////////////////
int Menu()
{
        int op;

        do
        {
                //system("cls");
```

```
                printf("\n\t ******* Administracion de Sucursales ********");
                printf("\n\t 1.Ingresar Sucursal");
                printf("\n\t 2.Borrar Sucursal");
                printf("\n\t 3.Borrar Cuenta");
                printf("\n\t 4.Visualizar Sucursales");
                printf("\n\t 5.Buscar Sucursal");
                printf("\n\t 6.Salir");

                printf("\n\t\n Teclee su opcion (1-6): ");
                scanf("%d", &op);
        } while(op < 1 || op > 6);

        return (op);
}

// Introducir un elemento en la lista nueva
void IngresarNuevaSucursal (struct Cuenta **pr, struct Cuenta **fi, char *nombre)
{
        struct Cuenta *pc, *fc, *q;
        int cant_cuentas = 0, i;

        pc = *pr;  // principio de la cola
        fc = *fi;  // final de la cola

        q = NuevoElemento ();
        strcpy(q->nom_suc, nombre);

        q->p_sig = NULL;

        /* Leer información sobre cuentas */

        printf("\n\t Cuantas cuentas posee esta sucursal?: ");
        scanf("%d",&cant_cuentas);
        q->cant_ctas = cant_cuentas;

        /* Asignar memoria para las cuentas */

        q->pIC = (Info_Cuenta *) malloc (cant_cuentas * sizeof (Info_Cuenta));

        if(q->pIC == NULL)
                Error();


        for (i = 0; i < cant_cuentas; i++)
        {
                fflush(stdin);

                printf("\n\t CUENTA No %d ", i + 1);
                printf("\n\t ----------------------");
```

```
                printf("\n\t Numero de cuenta: ");
                gets(q->pIC[i].No_cta);

                printf("\n\t Saldo: ");
                scanf("%f", &q->pIC[i].saldo);

                q->pIC[i].estado_reg = 'o'; // Registro lleno
                printf ("\n");
        }

        if (fc == NULL)
                pc = fc = q;

        else
        {
                fc->p_sig = q;
                fc = fc->p_sig;
        }

        *pr = pc;
        *fi = fc;
}

/* Crear un nuevo elemento */
struct Cuenta *NuevoElemento()
{
        return ((struct Cuenta *) malloc (sizeof (struct Cuenta) ) );
}

/* Función Error */
void Error (void)
{
        perror ("Error: insuficiente espacio de memoria.\n");
        exit (1);
}

void Visualizar(struct Cuenta *cabecera)
{
        struct Cuenta *actual = cabecera;
        int i = 0;
        int num_reg = 0;
        if (ListaVacia)
        {
                printf("Lista Vacia \n");
                return;
        }

        else
```

```
        {
                while (actual != NULL)
                {
                        printf ("\n\t REG %d \t", num_reg);
                        printf("%s  ",actual->nom_suc);
                        for (i = 0; i < actual->cant_ctas; i++)
                                printf("(%4s   ,   %4.2f);   ",   actual->pIC[i].No_cta,   actual-
>pIC[i].saldo);

                        printf (" %c ", 193);   // fin del archivo
                        printf ("\n");
                        actual = actual->p_sig;
                        num_reg++;
                }

                printf("\n");
        }
}


/********** Eliminar Lagunas de Memoria ******************/
void EliminarLagunas (struct Cuenta **principio)
{
        struct Cuenta *pl, *aux;

        pl = *principio;  // principio de la cola

        aux = pl;

        if (pl == NULL)
                return;   // Si la cola está vacía, retornar

        else
        {
                // Recorrer la lista desde el principio y eliminar las Sucursales
                while (pl != NULL)
                {
                        aux = pl;
                        pl = pl->p_sig;         // Avanzamos al siguiente

                        /* Primero liberamos el arreglo asignado para las cuentas */
                        free (aux->pIC);
                        aux->pIC = NULL;
                        /* y después liberamos la memoria asignado al elemento
                           apuntado por aux */
                        free (aux);
                }

        }  // Fin del else
```

```
        pl = aux = NULL;

        /* Actualizar los cambios */
        *principio = pl;

} // Fin de EliminarLagunas()


void BorrarCuenta(struct Cuenta **p, char *nom_suc, char *No_cuenta)
{
        struct Cuenta *aux;
        aux = *p;
        int i;
        bool NoCta_encontrada = false;          // registro encontrado?
        bool sucursal_encontrada = false;       // sucursal encontrada?

        while (aux != NULL) // Buscamos la sucursal
        {
                if ( strcmp(nom_suc, aux->nom_suc) == 0)
                {
                        sucursal_encontrada = true;
                        for (i = 0; i < aux->cant_ctas; i++) //Buscamos el No de cuenta
                        {
                                if ( strcmp(No_cuenta, aux->pIC[i].No_cta) == 0)
                                {
                                        strcpy(aux->pIC[i].No_cta, "");
                                        aux->pIC[i].saldo = 0.0;
                                        aux->pIC[i].estado_reg = 'b';
                                        printf("\n\t !! REGISTRO BORRADO !!");
                                        system ("pause");
                                        printf("\n");
                                        NoCta_encontrada = true;
                                        break;
                                }
                        } // Fin de for

                        break;
                } // Fin del if
                else
                        aux = aux->p_sig;     // Avanzar al siguiente

        } // Cierre del while

        // Verificamos si se han encontrado
        if (sucursal_encontrada == false)
        {
                printf("\n\t !!! SUCURSAL NO ENCONTRADA !!!!");
                system ("pause");
```

```
                printf("\n");
                return;
        }

        else    // Sucursal encontrada
        {
                if (NoCta_encontrada == false)
                {
                        printf("\n\t !!! SUCURSAL ENCONTRADA  Y CUENTA NO
ENCONTRADA !!!!");
                        system ("pause");
                        printf("\n");
                        return;

                }
        }

        return;
} // Fin de BorrarCuenta


void BorrarSucursal(struct Cuenta **p, struct Cuenta **f, char *nom_suc)
{
        struct Cuenta *cabecera = *p;
        struct Cuenta *actual = *p, *anterior = *p;
        if (ListaVacia)
        {
                printf("\n\t LISTA VACIA");
                return;
        }

        /* Entrar en la lista y encontrar el elemento a Borrar */
        while (actual != NULL)
        {
                if ( strcmp(actual->nom_suc, nom_suc) == 0 )
                {
                        // Dos posibles casos:
                        // 1. Borrar el elemento de inicio de la lista
                        if (actual == anterior)
                        {
                                if (*p == *f)    // Si solo hay un elemento
                                {
                                        *p = *f = actual->p_sig; // p = f = NULL
                                        free(actual->pIC);
                                        free (actual);
                                        printf ("\n\t REGISTRO BORRADO\n");
                                        system ("pause");
                                        return;
                                }
```

```
        // Si hay dos elementos
        if ((actual->p_sig)->p_sig == NULL )
        {
                *p = *f; // p = f
                free(actual->pIC);
                free (actual);
                printf ("\n\t REGISTRO BORRADO\n");
                system ("pause");
                return;
        }

        *p = actual->p_sig;

        free(actual->pIC);
        free (actual);
        printf ("\n\t REGISTRO BORRADO\n");
        system ("pause");
        return;
} // Fin de if(actual == anterior)

// 2. Borrar cualquier otro elemento, incluye el ultimo
else
{
        // Borrar un elemento que no es el ultimo
        if (actual->p_sig != NULL)
        {
                anterior->p_sig = actual->p_sig;
                free(actual->pIC);
                free(actual);
                printf ("\n\t REGISTRO BORRADO\n");
                system ("pause");

                return;
        }

        // Borrar el ultimo
        if(actual->p_sig == NULL)
        {
                anterior->p_sig = actual->p_sig;
                free(actual->pIC);
                free(actual);
                printf ("\n\t REGISTRO BORRADO\n");
                system ("pause");

                *f = anterior;
                return;
```

```
                        }
                    }
                } // Fin de if ( strcmp(actual->nom_suc, nom_suc) == 0 )

                anterior = actual;
                actual = actual->p_sig;
        } // Cierre del while

        if (actual == NULL)
                printf ("\n\t REGISTRO NO ENCONTRADO");

        else
                printf ("\n\t REGISTRO BORRADO");
}


/* Grabar la lista en el disco */
void GrabarEnDisco (struct Cuenta *cab)
{
        int i = 0;
        struct Cuenta *aux;

        pf = fopen("datos", "wb");    // abrir el fichero para escribir

        // rewind (puntero_fich);

        aux = cab;  // aux apunta a la cabecera de la lista

        // Verificar si la lista está vacía
        if (aux == NULL)
        {
                printf ("\n\t LISTA VACIA!!!");
                return;
        }

        // La lista tiene al menos un elemento
        // Recorrer la lista y grabar en el fichero referenciado por pf
        // cada elemento de la lista
        while (aux != NULL)
        {
                // Escribo la sucursal al disco
                fwrite (aux, tam_reg, 1, pf);

                // Escribo además, las cuentas para cada sucursal
                for (i = 0; i < aux->cant_ctas; i++)
                        fwrite (&aux->pIC[i], tam_InfoCuenta, 1, pf);

                aux = aux->p_sig;
        }
```

```
        // Cerrar el archivo
        fclose (pf);

} // fin de GrabarEnDisco()

void IntroducirListaNueva (struct Cuenta **pr, struct Cuenta**fi, struct Cuenta aux)
{
        struct Cuenta *pc, *fc, *q;
        int cant_cuentas = 0, i;
        Info_Cuenta aux_Info_Cuenta;
        int cant_leidos = 0;

        pc = *pr;  // principio de la cola
        fc = *fi;  // final de la cola

        q = NuevoElemento ();
        strcpy(q->nom_suc, aux.nom_suc);
        q->p_sig = NULL;
        q->cant_ctas = aux.cant_ctas;

        /* Asignar memoria para las cuentas */
        q->pIC = (Info_Cuenta *) malloc (q->cant_ctas * sizeof (Info_Cuenta));

        if(q->pIC == NULL)
                Error();

        // Ahora se leen las cuentas del disco pertenecientes a esta sucursal
        for (i = 0; i < q->cant_ctas; i++)
        {
                // Leer del disco
                cant_leidos = fread (&aux_Info_Cuenta, tam_InfoCuenta, 1, pf);
                bytes_totales += tam_InfoCuenta;

                if (ferror (pf) || feof (pf))
                {
                        perror ("Error al leer del fichero...\n");
                        exit(0);
                }
                // y guardarla en q
                q->pIC[i] = aux_Info_Cuenta;
        }

        if (fc == NULL)
                pc = fc = q;

        else
        {
                fc->p_sig = q;
```

```
                fc = fc->p_sig;
        }

        *pr = pc;
        *fi = fc;
}


void BuscarSucursal (struct Cuenta *cab, char *ns)
{
        struct Cuenta *rec_lista;
        rec_lista = cab;

        while (rec_lista != NULL)
        {
                if ( strcmp(ns, rec_lista->nom_suc) == 0)
                {
                        printf ("\n\t REGISTRO ENCONTRADO");
                        printf ("\n\t Nombre de la sucursal: %s", rec_lista->nom_suc);
                        printf ("\n\t Cantidad de cuentas: %d\n", rec_lista->cant_ctas);
                        return;
                }

                else
                        rec_lista = rec_lista->p_sig;

        } //fin del while

        // Si ha llegado aquí, es porque rec_lista ha acabado de recorrer la
        // lista, y ahora apunta a NULL
        if (rec_lista == NULL)
        {
                printf ("\n\t REGISTRO NO ENCONTRADO");
                return;
        }

}
```

Algoritmos utilizados en las principales funciones

Función IngresarNuevaSucursal

Function IngresarNuevaSucursal(struct Cuenta **pr, struct Cuenta **fi, char *nombre)
void
var *fc, *pc, *q: struct Cuenta
pc ← pr
fc ← fi

Reservar memoria para un elemento de tipo struct Cuenta, y referenciarlo por q:

q ← ReservarMemoria ();

Copiar el nombre de la sucursal en el campo correspondiente de q:
q->nom_suc ← nombre

Asignar a NULL el puntero p_sig de q: q->p_isg ← NULL

Pedir cantidad de cuentas: n ← cantidad de cuentas

Asignar memoria para n cuentas y dejar referenciado dicho espacio mediante el puntero pIC de q.

Para i ← 0, hasta n, hacer
    Pedir información para la cuenta i, y almacenarla en la posición i del arreglo.

Reasignar los punteros del inicio y final de la cola

Función BorrarSucursal

```
Function BorrarSucursal(struct Cuenta **p, struct Cuenta **f, char *nom_suc) void
var *cabecera ← *p, struct Cuenta
var *actual← *p, *anterior← *p, struct Cuenta
    Si ListaVacia
        Imprimir LISTA VACIA
        Regresar
    FinSi
    [Entrar en la lista y encontrar el elemento a Borrar]
    Mientras actual <> NULL
        Si actual->nom_suc= nom_suc
        [Dos posibles casos:
        1. Borrar el elemento de inicio de la lista]
            Si actual = anterior
                Si *p = *f      [Si solo hay un elemento]
                    *p ← *f ← actual->p_sig     [p ← f ← NULL]
                    Liberar actual->pIC
                    Liberar actual
                    Imprimir !! REGISTRO BORRADO!!
                    Regresar
                FinSi
                [Si hay dos elementos]
                Si actual->p_sig->p_sig = NULL
                    *p ← *f         [p ← f]
                    Liberar actual->pIC
                    Liberar actual
                    Imprimir !! REGISTRO BORRADO !!
                    Regresar
                FinSi
                *p ← actual->p_sig
                Liberar actual->pIC
```

```
                        Liberar actual
                        Imprimir !! REGISTRO BORRADO\n !!
                        Regresar
                FinSi actual = anterior
                [2.Borrar cualquier otro elemento, incluye el ultimo]
                SiNo
                        [Borrar un elemento que no es el último]
                        Si actual->p_sig <> NULL
                                anterior->p_sig ← actual->p_sig
                                Liberar actual->pIC
                                Liberar actual
                                Imprimir !! REGISTRO BORRADO !!
                                Regresar
                        FinSi
                        [Borrar el último]
                        Si actual->p_sig = NULL
                                anterior->p_sig ← actual->p_sig
                                Liberar actual->pIC
                                Liberar actual
                                Imprimir !! REGISTRO BORRADO\n !!
                                *f ← anterior
                                Regresar
                        FinSi
                FinSiNo
        FinSi actual->nom_suc, nom_suc ← 0
        anterior ← actual
        actual ← actual->p_sig
FinMientras

Si actual = NULL
        Imprimir !! REGISTRO NO ENCONTRADO !!
SiNo
        Imprimir !! REGISTRO BORRADO !!
FinSi

FinFunction
```

Función BorrarCuenta

```
Function BorrarCuenta(struct Cuenta **p, char *nom_suc, char *No_cuenta) void

var *aux: struct Cuenta
var CtaEncontrada, SucEncontrada: bool

aux ← principio de la lista de sucursales

Mientras aux <> NULL, hacer
        Si nom_suc = aux->nom_suc, entonces
                SucEncontrada ← true
```

```
            Para i ← 0 hasta cant_ctas de la sucursal
                Si No_cuenta = aux->pIC(i).No_cta
                    Poner valores vacíos en los campos de dicha cuenta
                    CtaEncontrada ← true
                FinSi
            FinPara
        FinSi
FinMientras
```

Función GrabarEnDisco

```
Function GrabarEnDisco (struct Cuenta *cab) void
var *aux: struct Cuenta

aux ← principio de la lista
Abrir fichero para escribir

Mientras aux <> NULL, hacer
        Escribir la sucursal referenciada por aux al disco.
        Para i ← 0 hasta la cantidad de cuentas de dicha sucursal
            Escribir la cuenta(i) al disco
        FinPara

        Avanzar el puntero aux para que apunte a la siguiente posición en la lista
FinMientras
```

6.2.2 Representación de registros de longitud variable mediante registros de longitud fija

Otra manera de implementar eficientemente los registros de longitud variable en un sistema de archivo, es utilizar unos o varios registros de longitud fija para representar cada registro de longitud variable.

Hay dos técnicas para hacer esto:

1) Método del espacio reservado:

En este caso se escoge una longitud máxima para todos los registros, y los espacios vacíos de los registros que no alcancen esa longitud se rellenan con un símbolo especial de valor nulo o de final de registro (⊥).

Ejemplo del espacio reservado:

Muestra el modo en que se representaría el archivo si se permitiera un máximo de tres cuentas por sucursal. Los registros de este archivo son del tipo lista-cuentas, pero el array contiene exactamente tres elementos, las sucursales con menos de tres cuentas tienen registros con campos con valores nulos. La Tabla 6-6, muestra lo expuesto.

Tabla 6-6: Ejemplo del método del espacio reservado

Registro	N_suc	Nº_cuenta	Saldo				
0	Centro	C-102	400	C-201	900	C-218	700
1	Mianus	C-305	350	⊥	⊥	⊥	⊥
2	Banpro	C-215	700	⊥	⊥	⊥	⊥
3	BDF	C-121	500	C-110	600	⊥	⊥
4	Becerril	C-222	700	⊥	⊥	⊥	⊥
5	Banco Uno	C-350	800	⊥	⊥	⊥	⊥

Problema:
Espacios no utilizados, se desperdicia una cantidad de espacio significativa.

Implementación de la técnica del espacio reservado en Lenguaje C

Estructura de datos utilizada para la representación del espacio reservado:

La estructura de datos y la algoritmia utilizada para construir este programa es muy similar a la representación en cadena de bytes, vista en el apartado anterior.

La única variación ahora será que, en lugar de preguntar al usuario la cantidad de cuentas que quiere introducir al momento de ingresar una nueva sucursal, ahora dicha cantidad de cuentas será fija y ya estará definida en el programa. En este caso particular, en nuestro programa se ha definido una cantidad de cuentas máxima de 4,

pudiendo ser cambiada en cualquier momento, sin alterar para nada el funcionamiento de nuestra aplicación.

Las definiciones de las estructuras de datos usadas para la programación de esta técnica, se muestra a continuación:

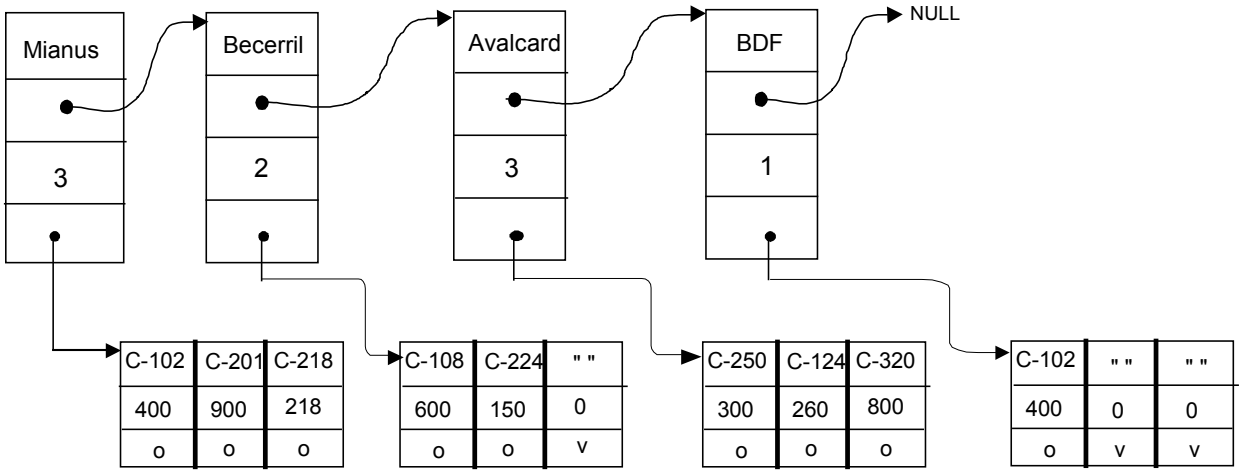
```
typedef struct
{
        char No_cta[20];      // Número de cuenta
        float saldo;          // Saldo de la cuenta
        char estado_reg;      // 'b' ->Borrado;'o' -> Lleno

}Info_Cuenta;


struct Cuenta
{
        char nom_suc [100];         // Nombre de la sucursal
        struct Cuenta *p_sig;       // Puntero al siguiente Elemento
        int cant_ctas;              // cantidad de cuentas
        Info_Cuenta *pIC;           // Puntero a Info_Cuenta
};
```

En la **Ilustración 6-34**, se muestra gráficamente, la estructura de datos con algunos valores de sucursales y de cuentas introducidos para esta técnica del espacio reservado.

Ilustración 6-34: Estructura de datos para la técnica del espacio reservado



A como se puede observar, todas las sucursales tienen una cantidad fija de cuentas (que se asignan dinámicamente en el momento de la ejecución del programa), en este caso 3; pero no todas dichas cuentas están ocupadas o conteniendo información útil de alguna cuenta. Las cuentas que no están siendo ocupadas, poseen en sus

campos, valores de cadena vacía para las cadenas de caracteres, y 0 para los campos enteros.

A continuación, presentaremos la ejecución del programa, mostrando la corrida de la diversas opciones del menú principal.

La **Ilustración 6-35**, muestra la ejecución inicial del programa:

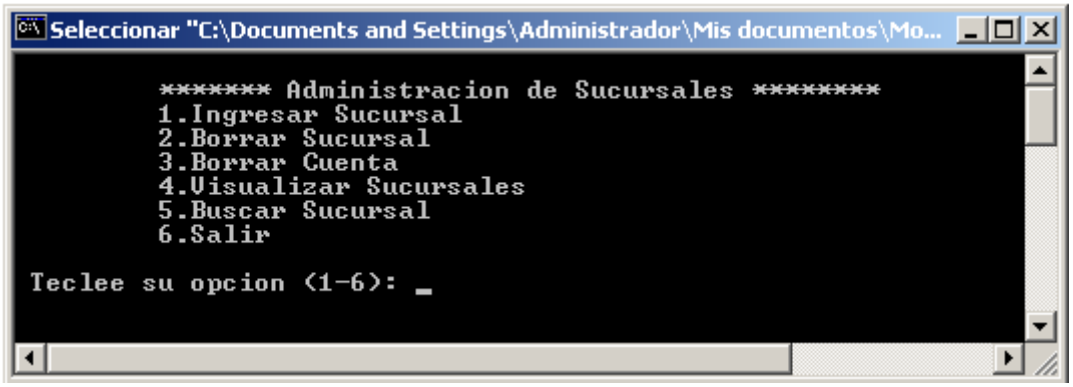


Ilustración 6-35: Menú inicial – Espacio reservado

Opción 1 – Ingresar Sucursal: Esta opción pide los datos para ingresar una nueva sucursal, y pide, además la información para un máximo de 4 cuentas, preguntando al finalizar de introducir una cuenta, si desea introducir otra. A como se pudo observar en la **Ilustración 6-34**, por cada nueva sucursal introducida, se crea un nuevo elemento que se inserta al final de la lista; se introduce el campo nombre de sucursal, y se actualiza el puntero psig; después, se asigna memoria para un array de 4 elementos de tipo Info_Cuenta, y se piden datos para las cuentas que el usuario quiera introducir. La **Ilustración 6-36**, muestra la introducción de una nueva sucursal –BDF – León, con 2 cuentas.

```
******* Administracion de Sucursales ********
1.Ingresar Sucursal
2.Borrar Sucursal
3.Borrar Cuenta
4.Visualizar Sucursales
5.Buscar Sucursal
6.Salir

Teclee su opcion (1-6): 1

Nombre de la sucursal: BDF-Leon

INTRODUZCA LA INFORMACION DE LAS CUENTAS (1 - 4)....
CUENTA No 1
------------------------
Numero de cuenta: C-101
Saldo: 400

Desea introducir otra cuenta (s/n): s
```

```
CUENTA No 2
-----------------------
Numero de cuenta: C-204

Saldo: 420


Desea introducir otra cuenta (s/n): n

******* Administracion de Sucursales ********
1.Ingresar Sucursal
2.Borrar Sucursal
3.Borrar Cuenta
4.Visualizar Sucursales
5.Buscar Sucursal
6.Salir

Teclee su opcion (1-6):
```

Ilustración 6-36: Introducción de una sucursal – Técnia del espacio reservado

Nota: Las entradas del usuario están marcadas en negrita, cursiva y subrayado.

Opción 2 – Borrar Sucursal:
Esta opción borra una sucursal y todas sus cuentas asociadas. Para ver la ejecución de esta opción, primero listaremos todas las sucursales que se han introducido con sus respectivas cuentas. La **Ilustración 6-37**, muestra dicho listado:

```
******* Administracion de Sucursales ********
1.Ingresar Sucursal
2.Borrar Sucursal
3.Borrar Cuenta
4.Visualizar Sucursales
5.Buscar Sucursal
6.Salir

Teclee su opcion (1-6): 4

REG 0   BDF-Leon  (C-101 , 400.00); (C-204 , 420.00); (     , 0.00); (     , 0.00);  ⊥

REG 1   Banpro-Chinandega  (C-208 , 680.00); (     , 0.00); (     , 0.00); (     , 0.00);  ⊥

REG 2   BAC-Managua  (C-226 , 420.00); (C-280 , 600.00); (C-128 , 680.00); (C-224 , 900.00);  ⊥

Presione una tecla para continuar . . .
```

Ilustración 6-37: Listado de sucursales – Técnica del espacio reservado

Un aspecto importante a señalar es que, la visualización de cada sucursal tiene el siguiente formato:

Número de Registro Nombre de la Sucursal (Num_Cta1, Saldo 1); (Num_Cta2, Saldo 2).....

A como se puede observar en la **Ilustración 6-37**, solamente la sucursal BAC – Managua, tiene ocupada sus cuatro cuentas, mientras que Banpro – Chinandega y BDF – León ocupan solamente 1 y 2 cuentas respectivamente.

La **Ilustración 6-38**, muestra el borrado de la sucursal Banpro – Chinandega. Después se listan de nuevo las sucursales para comprobar que la sucursal ha sido borrada.

```
       ******* Administracion de Sucursales ********
       1.Ingresar Sucursal
       2.Borrar Sucursal
       3.Borrar Cuenta
       4.Visualizar Sucursales
       5.Buscar Sucursal
       6.Salir

 Teclee su opcion (1-6): 2

       Nombre de la Sucursal: Banpro-Chinandega

       REGISTRO BORRADO
Presione una tecla para continuar . . .

       ******* Administracion de Sucursales ********
       1.Ingresar Sucursal
       2.Borrar Sucursal
       3.Borrar Cuenta
       4.Visualizar Sucursales
       5.Buscar Sucursal
       6.Salir

 Teclee su opcion (1-6): 4

REG 0   BDF-Leon  (C-101 , 400.00); (C-204 , 420.00); (     , 0.00); (     , 0.00); ⊥

REG 1   BAC-Managua  (C-226 , 420.00); (C-280 , 600.00); (C-128 , 680.00); (C-224 , 900.00); ⊥

Presione una tecla para continuar . . .
```

Ilustración 6-38: Borrado de una sucural – Técnica del espacio reservado

Opción 3 – Borrar cuenta: Esta opción borra una cuenta de una sucursal dada por el usuario. Nuestro programa lo que en realidad hace, es poner valores nulos en el lugar dentro del arreglo correspondiente a dicha cuenta.

De acuerdo a los datos resultantes mostrados en la **Ilustración 6-38**, la **Ilustración 6-39** muestra el borrado de la cuenta **C-280**, de la sucursal **BAC – Managua**.

```
       ******* Administracion de Sucursales ********
       1.Ingresar Sucursal
       2.Borrar Sucursal
       3.Borrar Cuenta
       4.Visualizar Sucursales
       5.Buscar Sucursal
       6.Salir

 Teclee su opcion (1-6): 3

       Nombre de la Sucursal: BAC-Managua

       No de la cuenta: C-280

       !! REGISTRO BORRADO !!Presione una tecla para continuar . . .


       ******* Administracion de Sucursales ********
       1.Ingresar Sucursal
       2.Borrar Sucursal
       3.Borrar Cuenta
       4.Visualizar Sucursales
       5.Buscar Sucursal
       6.Salir

 Teclee su opcion (1-6): 4

REG 0   BDF-Leon  (C-101 , 400.00); (C-204 , 420.00); (     , 0.00); (     , 0.00);  ⊥


REG 1   BAC-Managua  (C-226 , 420.00); (     , 0.00); (C-128 , 680.00); (C-224 , 900.00);  ⊥

Presione una tecla para continuar . . .
```

Ilustración 6-39: Borrado de una cuenta – Técnica del espacio reservado

Opción 4 – Visualizar sucursales

Esta opción ha sido ejecutada ya en varias ocasiones anteriores, por lo cual no será ejecutada de nuevo.

Opción 4 – Buscar Sucursal

Esta opción busca una sucursal por el nombre de dicha sucursal. Una vez que se encuentra, se imprime la información correspondiente al nombre y a la cantidad de cuentas. Si la sucursal no se encuentra, se imprime un rótulo de sucursal no encontrada. La **Ilustración 6-40**, muestra la búsqueda de la sucursal **BAC-Managua**.

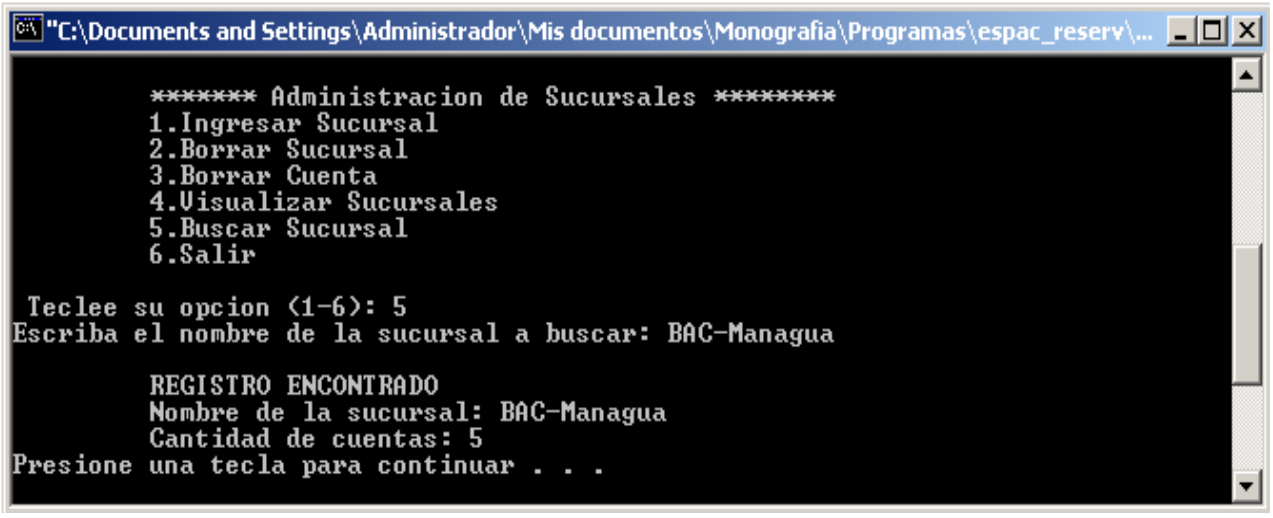


Ilustración 6-40: Buscar sucursal – Técnica del espacio reservado

Opción 6 – Salir

Esta opción hace terminar la ejecución de nuestro programa y aprovechamos para guardar la lista de sucursales con sus respectivas cuentas al disco duro. La **Ilustración 6-41**, muestra la terminación de nuestro programa y el mensaje de datos copiados al disco.

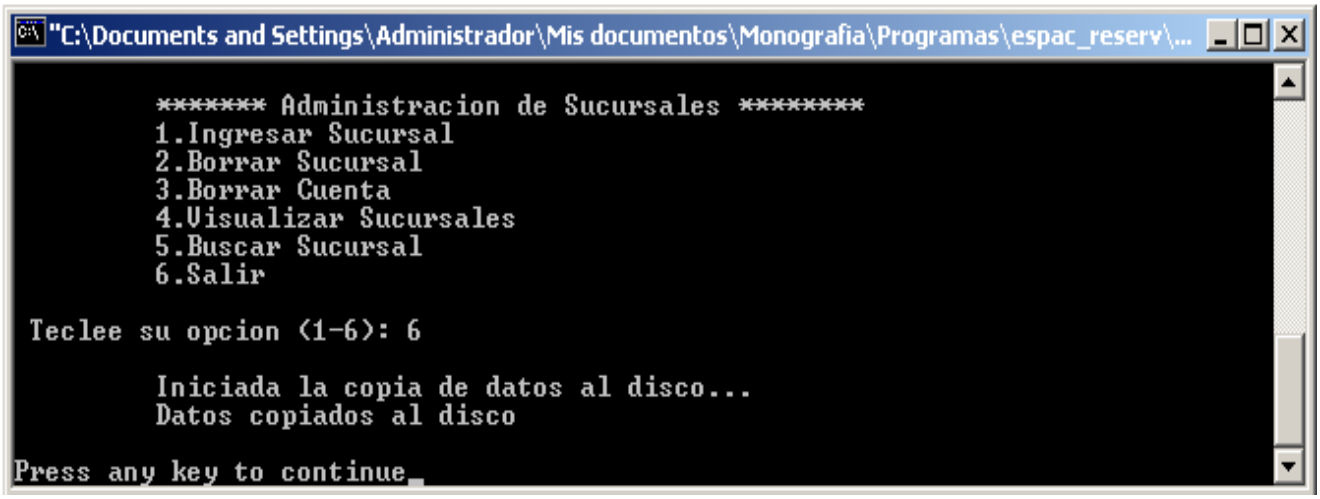


Ilustración 6-41: Terminación del programa – Técnica del espacio reservado

/* Principal.c */

```
/* Este programa implementa la representación de registros de longitud variable,
   mediante la técnica de longitud fija.
   Método del ESPACIO RESERVADO.
*/

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <crtdbg.h>
#include "definiciones.h"


// Función principal
void main()
{
	int opcion;
	int cant_leidos;
	struct Cuenta *p = NULL, *f = NULL;	// Principio y final de la cola
	char nom_suc[100];
	char No_cuenta[20];
	struct Cuenta aux;


	pf = fopen("datos", "rb");  // abrir el fichero datos para leer

	if (pf != NULL)
	{
		// El fichero existe, reconstruir la lista en MP
		// Traer datos del disco
		printf ("\n\t Trayendo datos del disco....");

		//rewind (pf);
		cant_leidos = fread (&aux, tam_reg, 1, pf);
		//bytes_totales += tam_reg;

		while (!feof (pf))
		{
			//fseek (pf, (long)bytes_totales, SEEK_SET);
			if (ferror(pf))
			{
				perror ("Error al leer del fichero\n");
				exit (0);
			}

			IntroducirListaNueva (&p, &f, aux);
```

```
                cant_leidos = fread (&aux, tam_reg, 1, pf);
                bytes_totales += tam_reg;

        }

        if (ferror(pf))
        {
                perror ("Error al leer del fichero");
                exit (0);
        }

        printf ("\n\t Datos traidos del disco....\n\n");

        fclose (pf);    // Cerramos el fichero

}

while (1)
{
        opcion = Menu();

        switch (opcion)
        {
                case 1:         // Ingresar nueva sucursal
                        fflush(stdin);
                        printf("\n\t Nombre de la sucursal: ");
                        gets(nom_suc);
                        IngresarNuevaSucursal (&p, &f, nom_suc);
                        break;

                case 2:
                        fflush(stdin);
                        printf("\n\t Nombre de la Sucursal: ");
                        gets(nom_suc);
                        BorrarSucursal(&p, &f, nom_suc);
                        break;

                case 3:         // Borrar Cuenta
                        fflush(stdin);
                        printf("\n\t Nombre de la Sucursal: ");
                        gets(nom_suc);

                        printf("\n\t No de la cuenta: ");
                        gets(No_cuenta);
                        BorrarCuenta (&p, nom_suc, No_cuenta);
                        break;

                case 4:
```

```
                    Visualizar(p);
                    system("pause");
                    printf("\n");
                    break;

                case 5:
                    fflush (stdin);
                    printf ("Escriba el nombre de la sucursal a buscar: ");
                    gets (nom_suc);
                    BuscarSucursal(p, nom_suc);
                    system("pause");
                    printf("\n");
                    break;


                case 6:
                    printf ("\n\t Iniciada la copia de datos al disco...");
                    GrabarEnDisco (p);
                    printf ("\n\t Datos copiados al disco \n\n");

                    exit(0);
                    break;


            } // fin switch

        } //fin del while

        EliminarLagunas(&p);

        if(_CrtDumpMemoryLeaks())
            printf("Hay lagunas de memoria");

} // fin del main
```

/* definiciones.h */

```
/* Definiciones de funciones */
#define ListaVacia (cabecera == NULL)
#define MAX_NUM_CTAS 4                  // 4 cuentas maximas por sucursal

// Funciones Prototipos
int Menu();
void IngresarNuevaSucursal(struct Cuenta **p, struct Cuenta **f, char *nom_suc);
struct Cuenta *NuevoElemento();
void Error (void);
void Visualizar (struct Cuenta *p);
void EliminarLagunas (struct Cuenta **pc);
void BorrarCuenta (struct Cuenta **p, char *nom_suc, char *No_cuenta);
void BorrarSucursal (struct Cuenta **p, struct Cuenta **f, char *nom_suc);
void GrabarEnDisco (struct Cuenta *cab);
void IntroducirListaNueva (struct Cuenta **p, struct Cuenta**f, struct Cuenta aux);
void BuscarSucursal (struct Cuenta *cab, char *ns);

typedef struct
{
        char No_cta[20];      // Número de cuenta
        float saldo;          // Saldo de la cuenta
        char estado_reg;      // 'b' ->Borrado;'o' -> Lleno

}Info_Cuenta;


struct Cuenta
{
        char nom_suc [100];         // Nombre de la sucursal
        struct Cuenta *p_sig;       // Puntero al siguiente Elemento
        int cant_ctas;              // Cantidad de cuentas
        Info_Cuenta *pIC;           // Puntero a Info_Cuenta
};

int tam_reg = sizeof (struct Cuenta);   // tamanyo de la estrutura datos
int tam_InfoCuenta = sizeof (Info_Cuenta);
int bytes_totales = 0;
FILE *pf;


////////////////////// Definición de Funciones ///////////////////
int Menu()
{
        int op;

        do
        {
```

```
                //system("cls");

                printf("\n\t ******* Administracion de Sucursales ********");
                printf("\n\t 1.Ingresar Sucursal");
                printf("\n\t 2.Borrar Sucursal");
                printf("\n\t 3.Borrar Cuenta");
                printf("\n\t 4.Visualizar Sucursales");
                printf("\n\t 5.Buscar Sucursal");
                printf("\n\t 6.Salir");

                printf("\n\t\n Teclee su opcion (1-6): ");
                scanf("%d", &op);
        } while(op < 1 || op > 6);

        return (op);
}

// Introducir un elemento en la lista nueva
void IngresarNuevaSucursal (struct Cuenta **pr, struct Cuenta **fi, char *nombre)
{
        struct Cuenta *pc, *fc, *q;
        char resp;
        int i, j;

        pc = *pr;  // principio de la cola
        fc = *fi;  // final de la cola

        q = NuevoElemento ();
        strcpy(q->nom_suc, nombre);

        q->p_sig = NULL;

        /* Asignar memoria para las cuentas */
        q->pIC = (Info_Cuenta *) malloc (MAX_NUM_CTAS * sizeof (Info_Cuenta));

        if(q->pIC == NULL)
                Error();

        /* Leer información sobre cuentas */
        printf ("\n\t INTRODUZCA LA INFORMACION DE LAS CUENTAS (1 - 4)....");

        for (i = 0; i < MAX_NUM_CTAS; i++)
        {
                fflush(stdin);

                printf("\n\t CUENTA No %d ", i + 1);
                printf("\n\t ----------------------");
```

```
        printf("\n\t Numero de cuenta: ");
        gets(q->pIC[i].No_cta);

        printf("\n\t Saldo: ");
        scanf("%f", &q->pIC[i].saldo);

        q->pIC[i].estado_reg = 'o'; // Registro lleno

        printf ("\n");

        do
        {
                fflush (stdin);
                printf ("\n\t Desea introducir otra cuenta (s/n): ");
                resp = getchar ();
        } while (resp != 's' && resp != 'n');

        if (resp == 'n')
                break;

        else
        {
                if (i == MAX_NUM_CTAS - 1)
                        printf ("\n\t Se ha agotado la cantidad de cuentas
                                disponibles...");
                continue;
        }

}

// Guardar en el campo cant_ctas la cantidas de cuentas introducidas
q->cant_ctas = i + 1;

// Inicializar el resto de los elementos a valores de cero o vacios
for (j = i + 1; j < MAX_NUM_CTAS; j++)
{
        strcpy (q->pIC[j].No_cta, "");
        q->pIC[j].saldo = 0.0;
        q->pIC[j].estado_reg = 'v';
}

if (fc == NULL)
        pc = fc = q;

else
{
        fc->p_sig = q;
        fc = fc->p_sig;
```

```
        }

        *pr = pc;
        *fi = fc;
}


/* Crear un nuevo elemento */
struct Cuenta *NuevoElemento()
{
        return ((struct Cuenta *) malloc (sizeof (struct Cuenta) ) );
}

/* Función Error */
void Error (void)
{
        perror ("Error: insuficiente espacio de memoria.\n");
        exit (1);
}


void Visualizar(struct Cuenta *cabecera)
{
        struct Cuenta *actual = cabecera;
        int i = 0;
        int num_reg = 0;
        if (ListaVacia)
        {
                printf("Lista Vacia \n");
                return;
        }

        else
        {
                while (actual != NULL)
                {
                        printf ("\nREG %d \t", num_reg);
                        printf("%s  ",actual->nom_suc);
                        for (i = 0; i < MAX_NUM_CTAS; i++)
                                printf("(%4s , %4.2f); ", actual->pIC[i].No_cta,
                                                        actual->pIC[i].saldo);

                        printf (" %c ", 193);   // fin del archivo
                        printf ("\n\n");
                        actual = actual->p_sig;
                        num_reg++;
                }
```

```
                printf("\n");
        }
}


/********** Eliminar Lagunas de Memoria ******************/
void EliminarLagunas (struct Cuenta **principio)
{
        struct Cuenta *pl, *aux;

        pl = *principio;  // principio de la cola

        aux = pl;

        if (pl == NULL)
                return;   // Si la cola está vacía, retornar

        else
        {
                // Recorrer la lista desde el principio y elminar los Clientes

                while (pl != NULL)
                {
                        aux = pl;
                        pl = pl->p_sig;         // Avanzamos al siguiente

                        /* Primero liberamos el arreglo asignado para las cuentas */
                        free (aux->pIC);
                        aux->pIC = NULL;

                        /* y después liberamos la memoria asignado al elemento
                           apuntado por aux */
                        free (aux);
                }

        }  // Fin del else

        pl = aux = NULL;

        /* Actualizar los cambios */

        *principio = pl;

} // Fin de EliminarLagunas()
```

```
void BorrarCuenta(struct Cuenta **p, char *nom_suc, char *No_cuenta)
{
	struct Cuenta *aux;
	aux = *p;
	int i;
	bool NoCta_encontrada = false;		// registro encontrado?
	bool sucursal_encontrada = false;	// sucursal encontrada?

	while (aux != NULL) // Buscamos la sucursal
	{
		if ( strcmp(nom_suc, aux->nom_suc) == 0)
		{
			sucursal_encontrada = true;
			for (i = 0; i < aux->cant_ctas; i++) //Buscamos el No de cuenta
			{
				if ( strcmp(No_cuenta, aux->pIC[i].No_cta) == 0)
				{
					strcpy(aux->pIC[i].No_cta, "");
					aux->pIC[i].saldo = 0.0;
					aux->pIC[i].estado_reg = 'b';
					printf("\n\t !! REGISTRO BORRADO !!");
					system ("pause");
					printf("\n");
					NoCta_encontrada = true;
					break;
				}
			} // Fin de for

			break;
		} // Fin del if

		else
			aux = aux->p_sig;	// Avanzar al siguiente
	} // Cierre del while

	// Verificamos si se han encontrado
	if (sucursal_encontrada == false)
	{
		printf("\n\t !!! SUCURSAL NO ENCONTRADA !!!!");
		system ("pause");
		printf("\n");
		return;
	}

	else	// Sucursal encontrada
	{
		if (NoCta_encontrada == false)
		{
```

```
                        printf("\n\t !!! SUCURSAL ENCONTRADA  Y CUENTA NO
                                ENCONTRADA !!!!");
                        system ("pause");
                        printf("\n");
                        return;

                }
        }

        return;
} // Fin de BorrarCuenta


void BorrarSucursal(struct Cuenta **p, struct Cuenta **f, char *nom_suc)
{
        struct Cuenta *cabecera = *p;
        struct Cuenta *actual = *p, *anterior = *p;
        if (ListaVacia)
        {
                printf("\n\t LISTA VACIA");
                return;
        }


        /* Entrar en la lista y encontrar el elemento a Borrar */
        while (actual != NULL)
        {
                if ( strcmp(actual->nom_suc, nom_suc) == 0 )
                {
                        // Dos posibles casos:
                        // 1. Borrar el elemento de inicio de la lista
                        if (actual == anterior)
                        {
                                if (*p == *f)     // Si solo hay un elemento
                                {
                                        *p = *f = actual->p_sig; // p = f = NULL
                                        free(actual->pIC);
                                        free (actual);
                                        printf ("\n\t REGISTRO BORRADO\n");
                                        system ("pause");
                                        return;
                                }


                                // Si hay dos elementos
                                if ((actual->p_sig)->p_sig == NULL )
                                {
                                        *p = *f; // p = f
```

```
                        free(actual->pIC);
                        free (actual);
                        printf ("\n\t REGISTRO BORRADO\n");
                        system ("pause");
                        return;
                }

                *p = actual->p_sig;

                free(actual->pIC);
                free (actual);
                printf ("\n\t REGISTRO BORRADO\n");
                system ("pause");
                return;
        } // Fin de if(actual == anterior)

        // 2. Borrar cualquier otro elemento, incluye el ultimo
        else
        {
                // Borrar un elemento que no es el ultimo
                if (actual->p_sig != NULL)
                {
                        anterior->p_sig = actual->p_sig;
                        free(actual->pIC);
                        free(actual);
                        printf ("\n\t REGISTRO BORRADO\n");
                        system ("pause");

                        return;
                }

                // Borrar el ultimo
                if(actual->p_sig == NULL)
                {
                        anterior->p_sig = actual->p_sig;
                        free(actual->pIC);
                        free(actual);
                        printf ("\n\t REGISTRO BORRADO\n");
                        system ("pause");

                        *f = anterior;
                        return;

                }
        }
} // Fin de if ( strcmp(actual->nom_suc, nom_suc) == 0 )

anterior = actual;
```

```
            actual = actual->p_sig;
        } // Cierre del while

        if (actual == NULL)
            printf ("\n\t REGISTRO NO ENCONTRADO");

        else
            printf ("\n\t REGISTRO BORRADO");

}

/* Grabar la lista en el disco */
void GrabarEnDisco (struct Cuenta *cab)
{
        int i = 0;
        struct Cuenta *aux;

        pf = fopen("datos", "wb");    // abrir el fichero para escribir

        // rewind (puntero_fich);

        aux = cab;  // aux apunta a la cabecera de la lista

        // Verificar si la lista está vacía
        if (aux == NULL)
        {
            printf ("\n\t LISTA VACIA!!!");
            return;
        }

        // La lista tiene al menos un elemento

        // Recorrer la lista y grabar en el fichero referenciado por pf
        // cada elemento de la lista
        while (aux != NULL)
        {
            // Escribo la sucursal al disco
            fwrite (aux, tam_reg, 1, pf);

            // Escribo además, las cuentas para cada sucursal
            for (i = 0; i < MAX_NUM_CTAS; i++)
                fwrite (&aux->pIC[i], tam_InfoCuenta, 1, pf);
            aux = aux->p_sig;
        }
        // Cerrar el archivo
        fclose (pf);

} // fin de GrabarEnDisco()
```

```
void IntroducirListaNueva (struct Cuenta **pr, struct Cuenta**fi, struct Cuenta aux)
{
	struct Cuenta *pc, *fc, *q;
	int cant_cuentas = 0, i;
	Info_Cuenta aux_Info_Cuenta;
	int cant_leidos = 0;

	pc = *pr;  // principio de la cola
	fc = *fi;  // final de la cola

	q = NuevoElemento ();
	strcpy(q->nom_suc, aux.nom_suc);
	q->p_sig = NULL;
	q->cant_ctas = aux.cant_ctas;

	/* Asignar memoria para las cuentas */
	q->pIC = (Info_Cuenta *) malloc (MAX_NUM_CTAS * sizeof (Info_Cuenta));

	if(q->pIC == NULL)
		Error();

	// Ahora se leen las cuentas del disco pertenecientes a esta sucursal
	for (i = 0; i < MAX_NUM_CTAS; i++)
	{
		// Leer del disco
		cant_leidos = fread (&aux_Info_Cuenta, tam_InfoCuenta, 1, pf);
		bytes_totales += tam_InfoCuenta;

		if (ferror (pf) || feof (pf))
		{
			perror ("Error al leer del fichero...\n");
			exit(0);
		}
		// y guardarla en q
		q->pIC[i] = aux_Info_Cuenta;
	}


	if (fc == NULL)
		pc = fc = q;

	else
	{
		fc->p_sig = q;
		fc = fc->p_sig;
	}

	*pr = pc;
```

```
        *fi = fc;
}

void BuscarSucursal (struct Cuenta *cab, char *ns)
{
        struct Cuenta *rec_lista;
        rec_lista = cab;

        while (rec_lista != NULL)
        {
                if ( strcmp(ns, rec_lista->nom_suc) == 0)
                {
                        printf ("\n\t REGISTRO ENCONTRADO");
                        printf ("\n\t Nombre de la sucursal: %s", rec_lista->nom_suc);
                        printf ("\n\t Cantidad de cuentas: %d\n", rec_lista->cant_ctas);
                        return;
                }

                else
                        rec_lista = rec_lista->p_sig;
        } //fin del while

        // Si ha llegado aquí, es porque rec_lista ha acabado de recorrer la
        // lista, y ahora apunta a NULL
        if (rec_lista == NULL)
        {
                printf ("\n\t REGISTRO NO ENCONTRADO");
                return;
        }
}
```

Como se ha dicho antes y como se ha podido observar, el código expuesto en esta técnica del espacio reservado, es muy similar al utilizado en la técnica de cadena de bytes. Por esta razón, el único algoritmo que se explicará es el de la función IngresarNuevaSucursal, que es el que varía más respecto al anterior.

Funcion IngresarNuevaSucursal

```
Function IngresarNuevaSucursal (struct Cuenta **pr, struct Cuenta **fi, char *nombre)
constante MAX_NUM_CTAS ← 4
*pc, *fc, *q, struct Cuenta
var resp, caracter
var i, j, integer
pc ←*pr [principio de la cola]
fc ←*fi[final de la cola]
q ← ReservarMemoria ()
q->nom_suc ← nombre
q->p_sig ← NULL

[Asignar memoria para las cuentas]
q← ReservarMemoria para MAX_NUM_CTAS
Si q->pIC = NULL
        Error()
FinSi
[Leer información sobre cuentas]
Imprimir INTRODUZCA LA INFORMACION DE LAS CUENTAS (1 - 4)
Para i ← 0, i < MAX_NUM_CTAS, i++
        Leer información para cuenta i
        Preguntar al usuario si desea introducir otra cuenta
        Si respuesta del usuario es afirmativa, entonces
                Si no se ha agotado MAX_NUM_CTAS, entonces
                        Regresar al ciclo
                SiNo
                        Imprimir !! CANTIDAD DE CUENTAS AGOTADAS !!
                FinSi
        SiNo
                Salir del ciclo
        FinSi
FinPara
[Guardar en el campo cant_ctas la cantidas de cuentas introducidas]
q->cant_ctas ← i + 1

[Inicializar el resto de los elementos a valores de cero o vacios]
Para j ← i + 1, j < MAX_NUM_CTAS, j++
        Escribir valores nulos en los campos del registro j
FinPara
```

```
Si fc = NULL
        pc ← fc ← q
SiNo
        fc->p_sig ← q
        fc ← fc->p_sig
FinSiNo

*pr ← pc
*fi ← fc
FinFunction
```

2) Representación con listas o métodos de punteros:

El registro de longitud variable se representa mediante una lista de registros de longitud fija, enlazada mediante punteros.

Ejemplo del método de punteros:

Como muestra la **Ilustración 6-42**, se enlazan todos los registros pertenecientes a la misma sucursal.

Registro	N-suc	N°_cuenta	Saldo
0	Banpro	C-102	80.000
1	Centro	C-305	70.000
2	BDF	C-101	140.000
3	Banic	C-222	200.000
4	Mianus	C-217	180.000
5		C-220	140.000
6	Becerril		
7		C-110	120.000
8		C-230	130.000

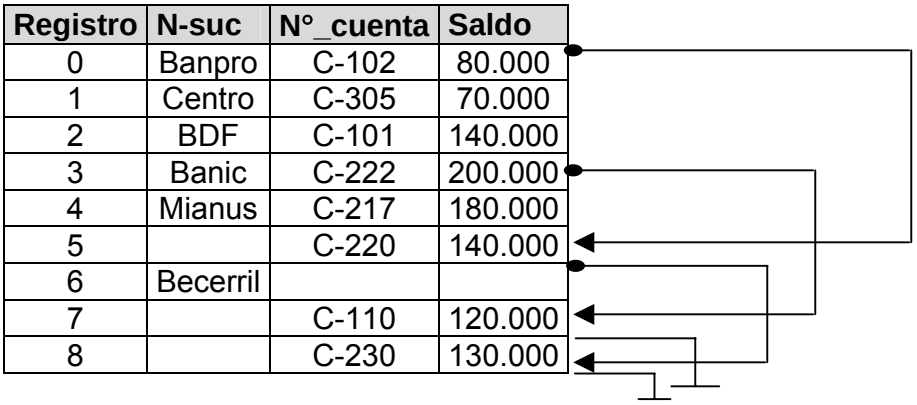


Ilustración 6-42: Esquema gráfico del método de punteros

Problema:

Se desperdicia espacio en todos los registros excepto en el primero de la serie. El primer registro debe tener el valor nombre_sucursal, pero los registros siguientes no necesitan tenerlo. No obstante, hay que incluir en todos los registros este campo o los registros no serán de longitud constante.

Para resolver este problema se permiten en el archivo dos tipos de bloques:

d) **Bloque Ancla:** que contiene el primer registro de cada cadena.

e) **Bloque de desbordamiento:** que contiene los registros que no son los primeros de sus cadenas.

Ejemplo: En la **Ilustración 6-43**, se muestra una propuesta de solución al problema planteado.

Bloque Ancla

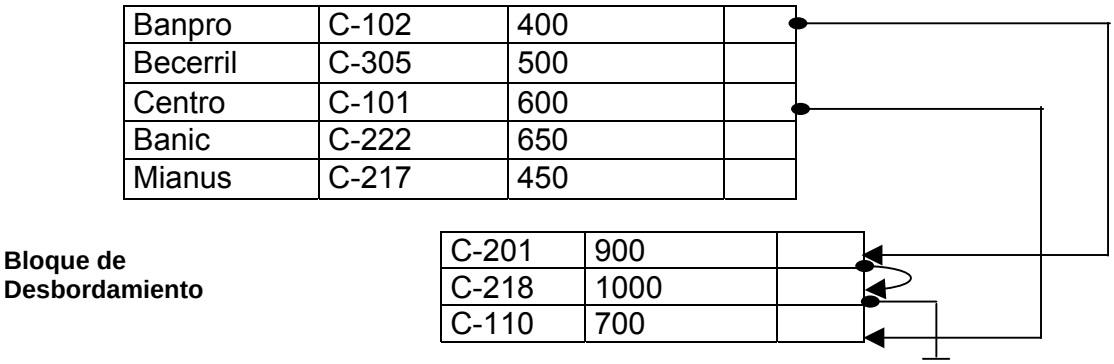


Ilustración 6-43: Esquema gráfico del bloque ancla

Ventaja:
Reducción del número de acceso a bloque.

Problema:
Al crecer el número de registro de un bloque, este se desborda (no caben más registros).

Solución:
Crear otro bloque y apuntarlo.

6.3 Organización de archivos secuenciales

Los archivos secuenciales están diseñados para el procesamiento eficiente de los registros de acuerdo a un órden basado en una clave de búsqueda.

Una clave de búsqueda es cualquier atributo o conjunto de atributos.

Para permitir la recuperación rápida de los registros según el orden de la clave de búsqueda, los registros se vinculan mediante punteros.

Para minimizar el número de accesos a los bloques en el procesamiento de los archivos secuenciales, los registros se guardan físicamente de acuerdo con el orden indicado por la clave de búsqueda.

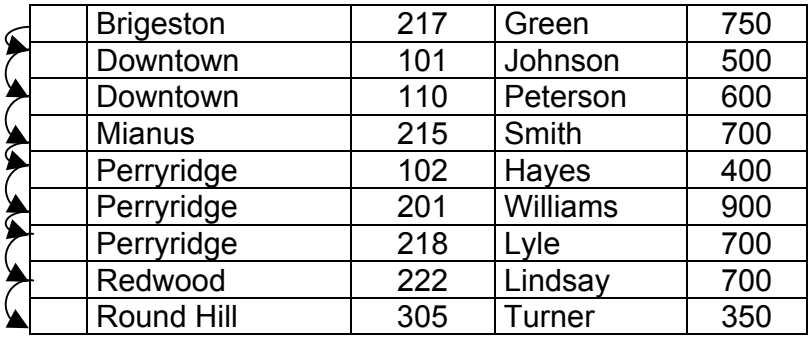
La organización secuencial de archivo permite que los registros se lean de forma ordenada, lo que puede ser útil para la visualización.

Resulta difícil mantener el orden físico secuencial cuando se insertan y borran registros, dado que resulta costoso desplazar muchos registros como consecuencia de una sola inserción o borrado. Se puede gestionar el borrado utilizando cadenas de punteros.

Para la inserción se aplican las siguientes reglas:
1. Localizar el registro del archivo que precede al registro que se va a insertar en el orden de la clave de búsqueda.

2. Si existe algún registro vacío (es decir, un espacio que haya quedado libre después de un borrado) dentro del mismo bloque que ese registro, el nuevo registro se insertará ahí. En caso contrario el nuevo registro se insertará en un bloque de desbordamiento. En cualquier caso, hay que ajustar los punteros para vincular los registros según el orden de la clave de búsqueda.

Ejemplo: Sea el archivo secuencial indicado en la **Ilustración 6-44**.



	Brigeston	217	Green	750
	Downtown	101	Johnson	500
	Downtown	110	Peterson	600
	Mianus	215	Smith	700
	Perryridge	102	Hayes	400
	Perryridge	201	Williams	900
	Perryridge	218	Lyle	700
	Redwood	222	Lindsay	700
	Round Hill	305	Turner	350

Ilustración 6-44: Archivo secuencial ordenado por el campo nombre

Ejemplo: En la Ilustración 6-45, se muestra el proceso para insertar un nuevo registro.

Ilustración 6-45: Archivo Secuencial al insertar el registro (North Town, 888, Adams, 800)

6.4 Indexación y asociación

6.4.1 Árboles

Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas. En un árbol existe un nodo especial denominado raíz. Un nodo del que sale alguna rama, recibe el nombre de nodo de bifurcación o nodo rama y un nodo que tiene ramas recibe el nombre de nodo terminal o nodo hoja. En la Ilustración 6-46, se muestra lo expuesto.

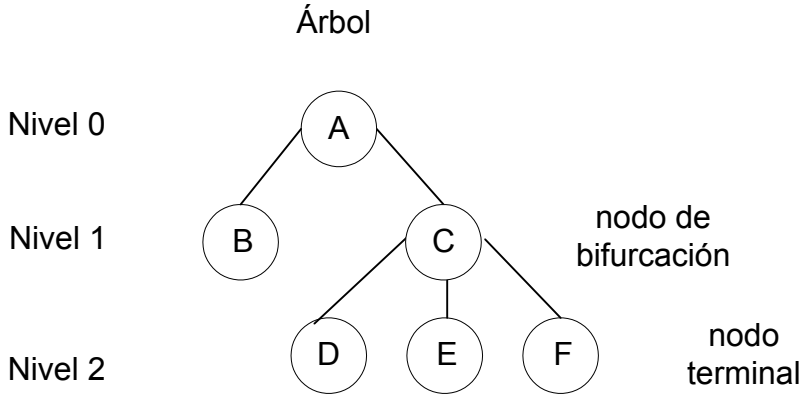


Ilustración 6-46: Estructura de un árbol

De un modo más formal, diremos que un árbol es un conjunto finito de uno o más nodos tales que:

a) Existe un nodo especial llamado raíz del árbol, y

b) Los nodos restantes están agrupados en n > 0 conjuntos disjuntos $\mathbf{A_1,\ldots,A_n}$, cada uno de los cuales es a su vez un árbol, que recibe el nombre del subárbol de la raíz.

Las formas de recorrer un árbol se muestran en la Ilustración 6-47.

Ilustración 6-47: Forma de recorrer un árbol

Donde **D**: derecha; **I**: izquierda; **R**: raiz.

6.4.2 Árboles binarios

Un árbol binario es un conjunto finito de nodos, el cual puede ser vacío o, puede contener un par de árboles binarios disjuntos, que son llamados subárbol izquierdo subárbol derecho.

Árboles de Búsqueda Binaria

Un árbol de búsqueda binario sobre la colección de n registro con llaves k_1, k_2, ..., k_n es un árbol binario, donde cada uno de sus nodos R_i contiene una de las llaves k_i, para i = 1, 2, , ..., n. Las llaves son los identificadores de los nodos. La búsqueda de un nodo particular se hará buscando su valor de llave.

Los nodos de un árbol de búsqueda binaria están arreglados de tal forma que la búsqueda de una de una llave en particular prosigue hacia abajo sobre alguna rama del árbol. El valor de la llave buscada es comparada contra el valor de la llave de la raíz del árbol: sí éste es menor que el valor de la raíz, la búsqueda prosigue hacia abajo por el subárbol izquierdo; si es mayor que el valor de la raíz, la búsqueda prosigue hacia abajo sobre el subárbol derecho. La misma lógica es aplicada en cada nodo encontrado, hasta satisfacer la búsqueda o hasta determinar que la llave buscada no está incluida en el árbol.

Por lo regular, el valor de una llave no está solo. Más bien, el valor de la llave está asociado con campos de información, para formar un registro. Almacenar estos campos de información en un árbol binario de búsqueda podría hacer muy grande al árbol. Para acelerar las búsquedas y reducir el tamaño del árbol, los campos de información de los registros normalmente quedan organizados en archivos y son almacenados por separado.

La conexión entre un valor de llave en el árbol binario de búsqueda y el registro respectivo en el archivo, se establece alojando un apuntador hacia el registro con el valor de la llave. Esto califica el árbol de búsqueda binario como un índice.

Ejemplo:

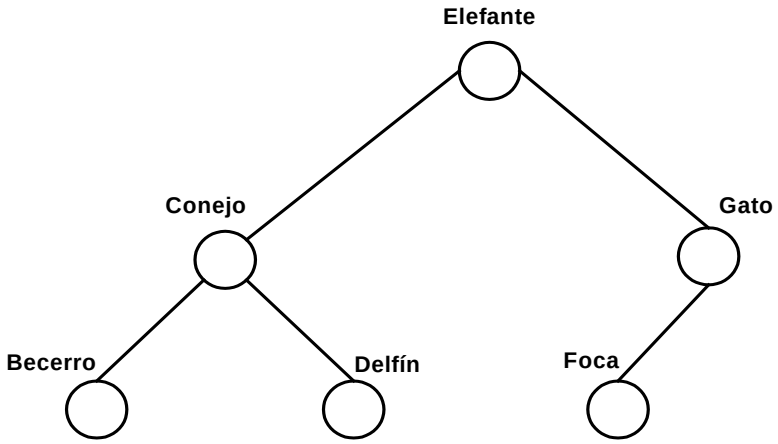


Ilustración 6-48: Un árbol de búsqueda binaria

En la Ilustración 6-49, se muestra el árbol de la Ilustración 6-48, pero ahora utilizado como índice.

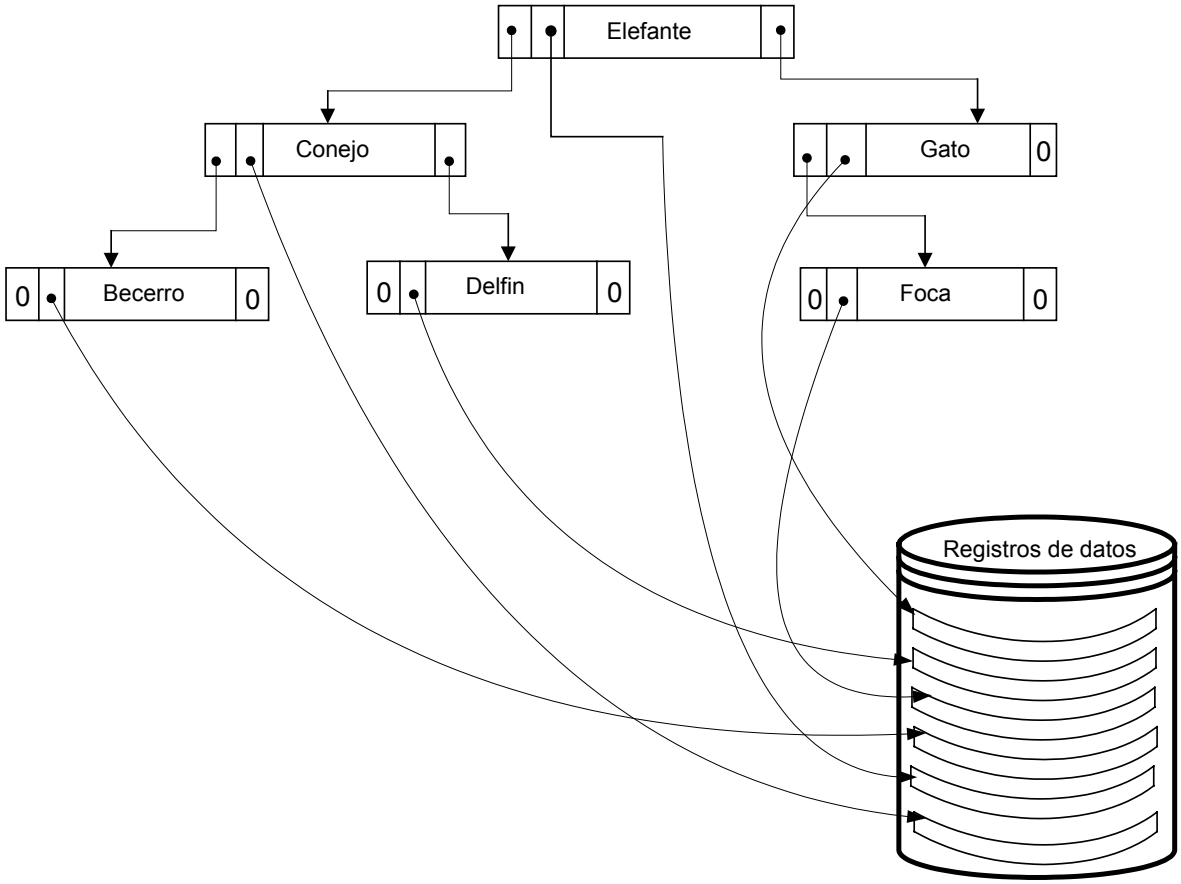


Ilustración 6-49: Árbol de búsqueda binaria de la Ilustración 6-48 , utilizado como índice.

Un índice es una colección estructurada de valores de llaves y direcciones, por pares; el propósito principal del índice es facilitar el acceso hacia una colección de registros. Un índice se dice que es un **índice denso** si contiene una pareja de valores de llave y dirección para cada registro de la colección. Un índice que no es denso es llamado algunas veces **índices dispersos**.

6.4.3 Árbol de Búsqueda de M-Vías.

Un árbol de búsqueda de M-vías es un árbol en el cual cada nodo tiene un grado externo <= m. Cuando un árbol de esta naturaleza no está vacío, tiene las siguientes propiedades:

1. Cada nodo del árbol tiene la estructura que muestra la Ilustración 6-50:

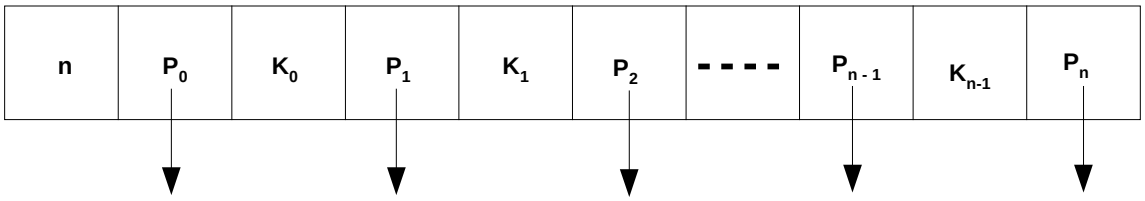


Ilustración 6-50: Estructura general de un árbol de m vías

Donde P_0, P_1, …, P_n son apuntadores a los nodos de los subárboles K_0, ..., K_{n-1} son los valores de llaves. El registro de que cada nodo tenga un grado externo <= m obliga a que n <= m-1.

2. Los valores de llaves en un nodo están en orden ascendente:

$$K_i < K_{i+1}$$
$$para \quad i = 0, \ldots, n-2$$

3. Todos los valores de las llaves que están en los nodos del subárbol apuntado por P_i, son menores al valor de la llave K_i para i = 0, ..., n -1.

4. Todos los valores de las llaves que están en los nodos del subárbol apuntado por P_n son mayores al valor de la llave K_{n-1}.

5. Los subárboles apuntados por P_i, i = 0, ..., n también son árboles de búsquedas de M-vías.

Ejemplo: A continuación se muestra un árbol de búsqueda de tres vías. Solamente son mostrados los valores de las llaves y los apuntadores de los subárboles no nulos, con un máximo de tres apuntadores de subárboles.

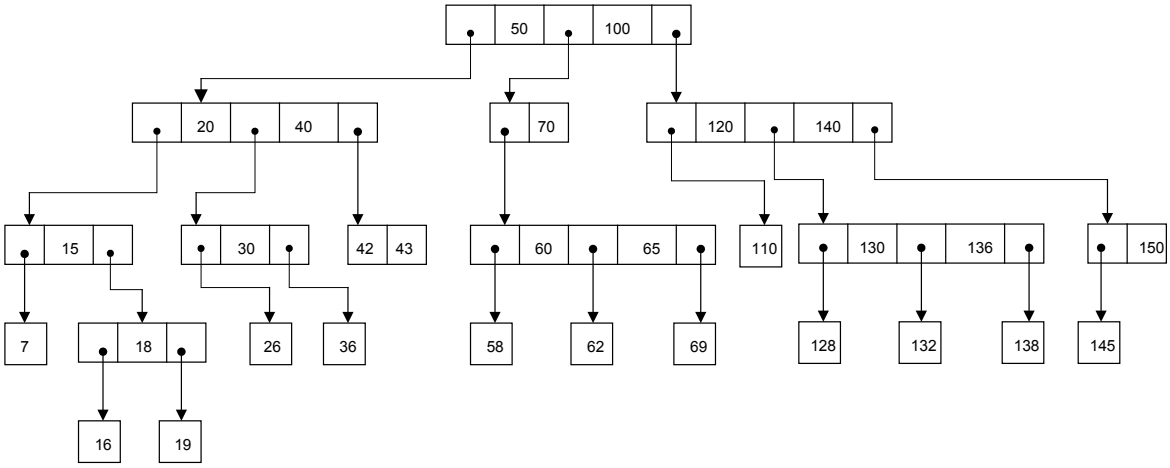


Ilustración 6-51: Ejemplo de un árbol de búsqueda de tres caminos

6.4.4 Árboles B

Un árbol-B de orden m es un árbol de búsqueda de m-vías con las siguientes propiedades:

1. Cada nodo del árbol, excepto la raíz y las hojas, tiene al menos (1/2m) subárboles y no más de m subárboles.

2. La raíz del árbol tiene al menos dos subárboles, a menos que él mismo sea una hoja.

3. Todas las hojas del árbol están en el mismo nivel.

La primera restricción a segura que cada nodo del árbol esté lleno al menos a la mitad. La segunda restricción obliga a que el árbol se ramifique pronto. La tercera restricción conserva el árbol casi balanceado.

Mantener un árbol-B es significativamente más fácil que mantener un árbol óptimo de búsqueda de m-vías. Las razones para usar un árbol-B de orden m en lugar de usar un árbol óptimo de búsqueda de m-vías, es que cuesta menos esfuerzo mantener la estructura de un árbol-B, cuando las llaves son borradas o insertadas; aún las longitudes máximas de búsqueda son casi tan buenas como las de una estructura óptima de un árbol de búsqueda de m-vías del mismo orden.

Inserción en un Árbol-B

La lógica para insertar un valor de llave en un árbol B es:

Primero, debe realizar una búsqueda en el árbol-B (que es la misma que la de un árbol de búsqueda de m-vías) para encontrar la localización correcta para el valor de llave, en ese árbol. Después se inserta la llave. Usualmente, con árboles de mayor orden, el nodo tiene espacio disponible para otro valor de llave y para un apuntador. Aunque algunas veces, el nodo ya está completo y no puede alojar otro valor de llave. En ese caso el nodo debe ser partido en dos, con la mitad de los valores de llaves y apuntadores hacia un nodo y la otra mitad hacia el otro nodo. Estos nodos “gemelos” tienen al mismo padre, el cual deberá ser modificado para incluir el valor de llave y el apuntador adicional. En el peor de los casos, el procedimiento de partición debe ser continuado a lo largo de toda la trayectoria del árbol hasta la raíz.

En la Ilustración 6-52, se muestra el algoritmo usado para insertar un elemento en un árbol B.

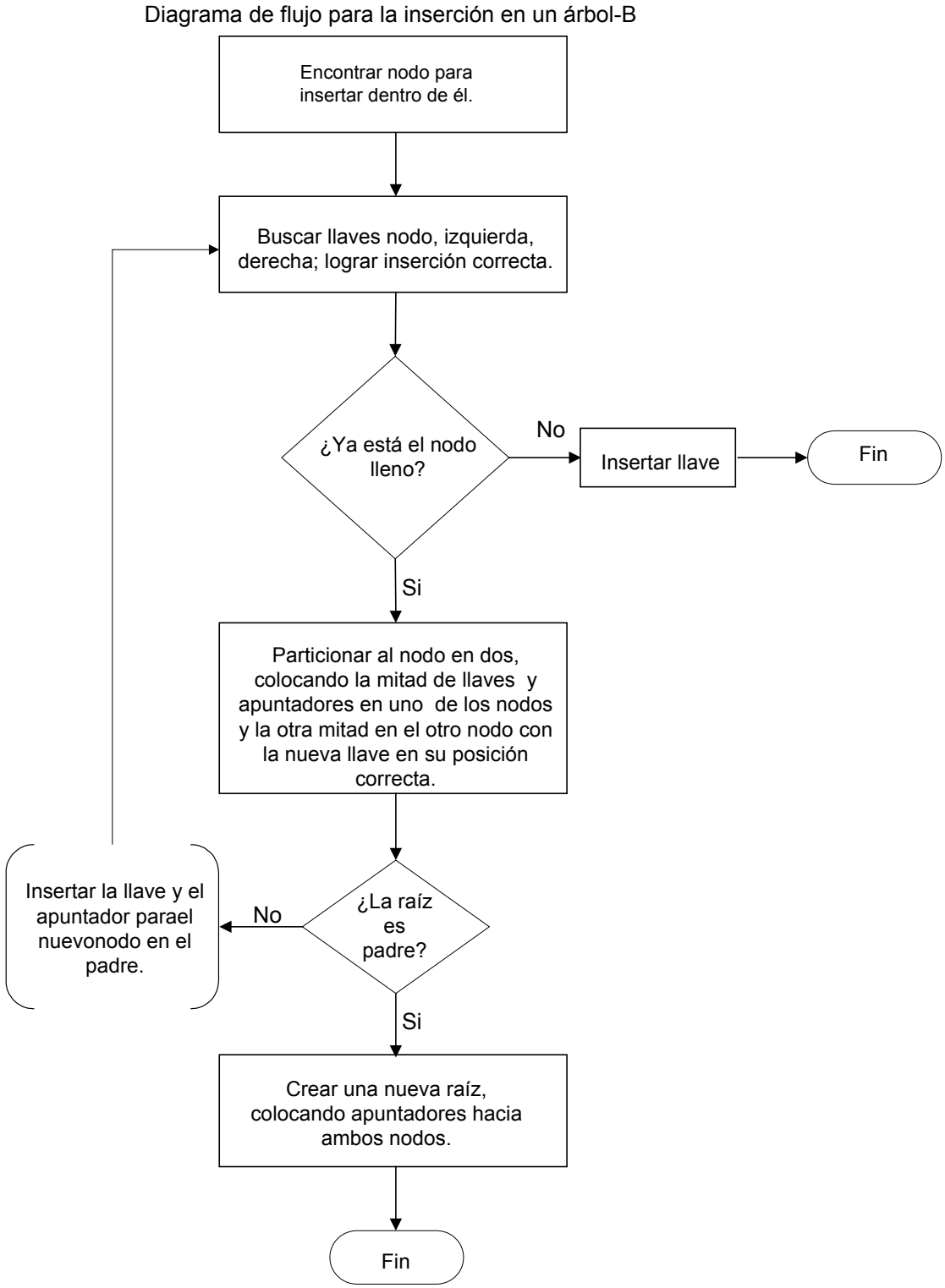


Ilustración 6-52: Diagrama de flujo para la inserción en un árbol B

Ejemplo 1:

En la Ilustración 6-53, se muestra un árbol-B de orden tres. Cada nodo ha sido marcado con una letra para facilitar posteriores referencias a ese árbol de ejemplo. Vamos a realizar la inserción de algunas llaves nuevas: 22, 41, 59, 57, 54, 33, 75, 124, 122, 123.

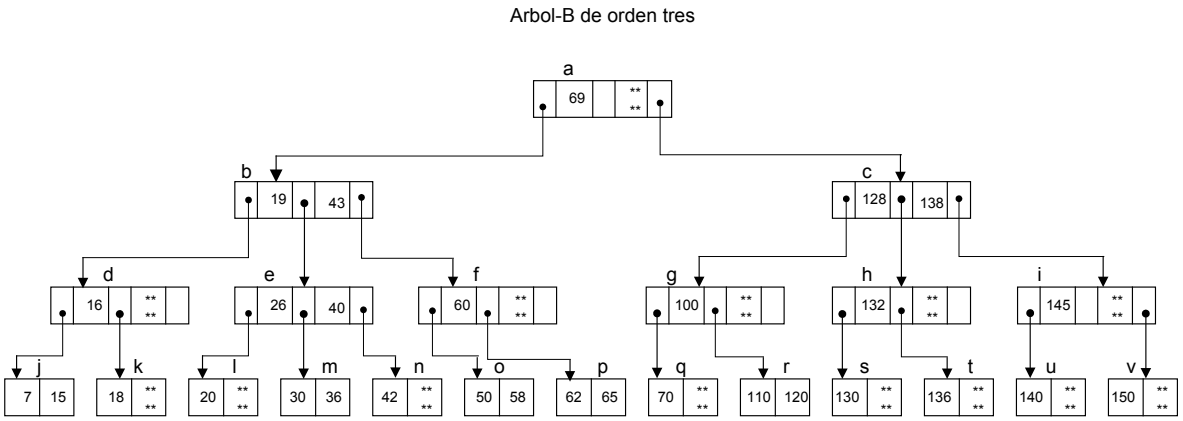


Ilustración 6-53: Árbol B del ejemplo 1

Inserción de la clave 22:

La llave con valor 22 deberá residir en el nodo L; este nodo tiene espacio suficiente para otra llave. El proceso se muestra en la Ilustración 6-54.

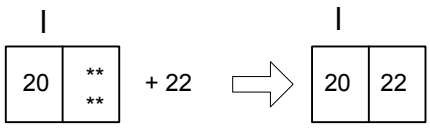


Ilustración 6-54: Inserción de la clave 22

Inserción de la clave 41:

De igual manera la llave con valor 41, la cual deberá residir en el nodo n, puede insertarse ahí porque existe espacio disponible. El proceso se muestra en la Ilustración 6-55

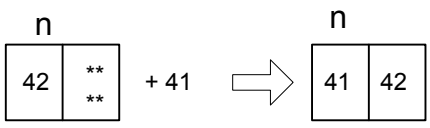


Ilustración 6-55: Inserción de la clave 41

Note que la secuencia adecuada de los valores de las llaves debe mantenerse; la llave con valor 42 es recorrida hacia la derecha para dar cabida a la llave con valor 41.

Inserción de la clave 59:

La llave con valor 59 deberá residir en el nodo o. Sin embargo, el nodo o ya contiene el máximo número de llaves permitido para un árbol-B de orden tres. Cada vez que una llave necesita ser insertada en un nodo que ya esta lleno, ocurre una partición. De un nodo se hacen dos nodos; la mitad de sus llaves van hacia uno de los dos nodos y la otra mitad al otro nodo. El valor restante es movido hacia el nodo padre. El nodo padre también tiene que ser ajustado para incluir un apuntador hacia el nodo engendrado. Aquí los nodos f y o son modificados. La Ilustración 6-56, muestra lo expuesto.

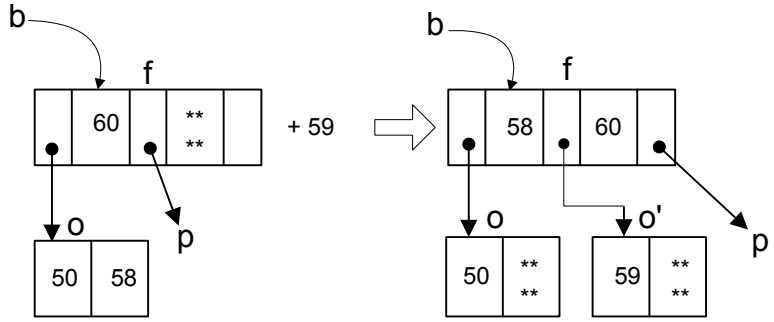


Ilustración 6-56: Inserción de la clave 59

Inserción de la clave 57:

Ahora, la inserción del 57, el siguiente valor de llave, prosigue simplemente guardando el 57 en el nodo o. El proceso se muestra en la Ilustración 6-57.

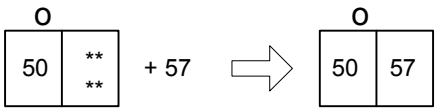


Ilustración 6-57: Inserción de la clave 57

Inserción de la clave 54:

La llave con valor 54, también debería ser guardada en el nodo o, pero el nodo o está de nuevo lleno. El mismo algoritmo de partición es aplicado nuevamente, como muestra la Ilustración 6-58.

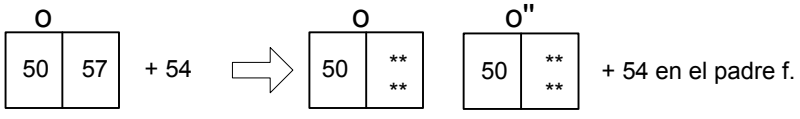


Ilustración 6-58: Inserción de la clave 54

Sin embargo, el nodo padre f también está lleno y no puede contener otro valor de llave y un apuntador al nodo o'' (o'': nodo resultante al hacer la inserción). El algoritmo de partición es aplicado nuevamente, esta vez al nodo f. Lo expuesto se muestra en la Ilustración 6-59.

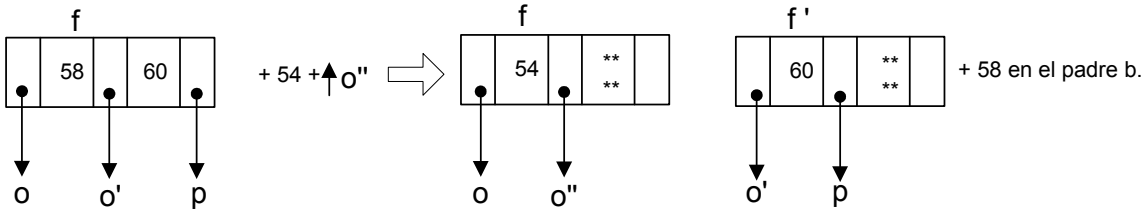


Ilustración 6-59: Algoritmo de partición aplicado al nodo f

De nuevo sucede que el nodo padre b está lleno y no puede acomodar otro valor de llave y el apuntador a f '. El procedimiento de partición es aplicado ahora sobre el nodo b. La Ilustración 6-60, muestra gráficamente lo expuesto

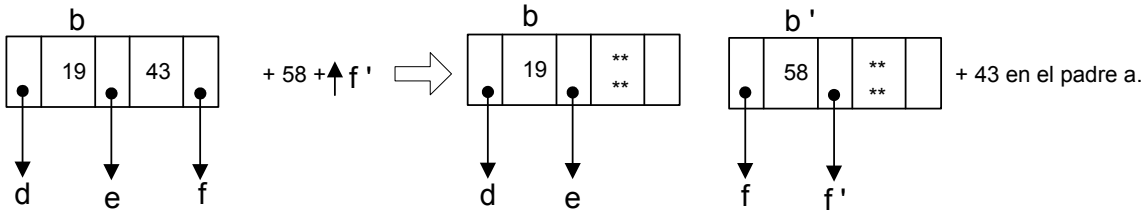


Ilustración 6-60: Algoritmo de partición aplicado al nodo b

El nodo padre **a** tiene espacio disponible para otro valor de llave y un apuntador a b'.

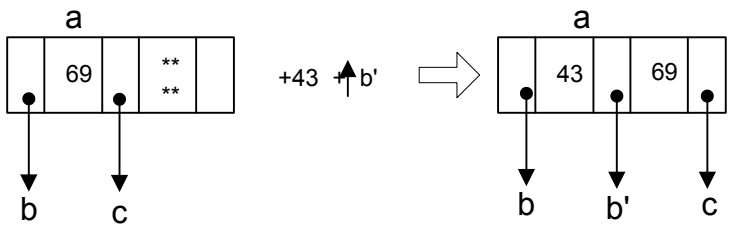


Ilustración 6-61: Finalizando la inserción de la clave 54

La inserción de las siguientes cuatro llaves 33, 75, 124, 122 puede acomodarse fácilmente .La inserción del 123, sin embargo, provoca que el árbol-B crezca un nivel más. También se realizan las inserciones 65, 7, 40, 16. El árbol resultante es mostrado en la Ilustración 6-62.

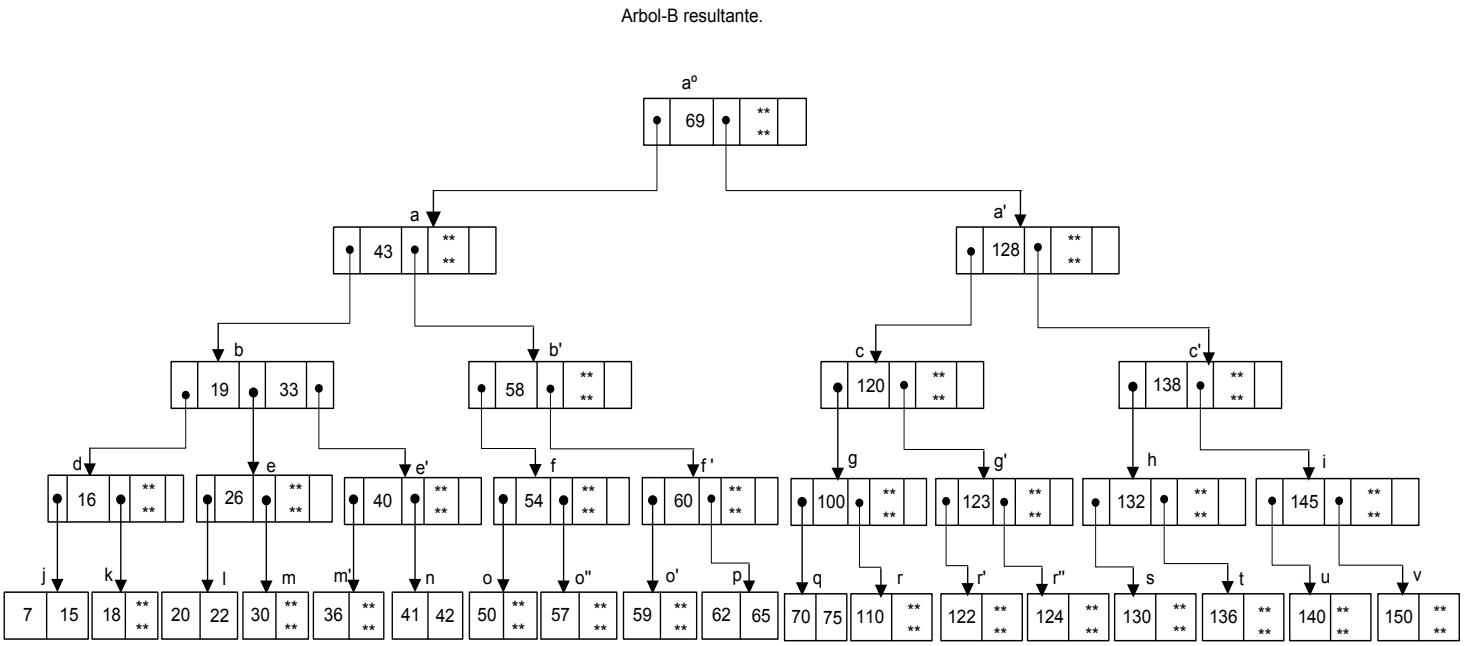


Ilustración 6-62: Árbol total resultante luego de insertar las claves: 22, 41, 59, 57, 54, 33, 75, 124, 122, 123, 65, 7, 40 y 16

Supresión de un Árbol-B

La supresión de una llave de un árbol-B es ligeramente más complicada que la inserción.

Para conservar un árbol-B como árbol-B, dos nodos necesitan ser combinados cuando un nodo tenga menos llaves y apuntadores del mínimo requerido.

Como en una inserción una búsqueda en el árbol –B debe hacerse primero para encontrar el nodo que contenga a la llave por suprimir.
Si la llave a suprimir está en un nodo interior, el árbol necesita ser transformado para mover esta llave hacia una hoja, pues ésta resulta más fácil de suprimir. Una vez hecha la supresión, otras llaves pueden tener que ser recorridas, o combinar hojas para satisfacer los requisitos de mínimos de llaves y apuntadores para un árbol–B. Estas combinaciones de nodos pueden ser necesarias en varios niveles.

Si una combinación se propaga a través de toda la trayectoria hacia el nivel superior, una nueva raíz se constituye y la altura del árbol-B es disminuida en 1. Lo mismo que el caso de la partición, la probabilidad de que ocurra una combinación de nodos decrece a medida que aumenta el orden del árbol-B.

En la Ilustración 6-63, se muestra el diagrama de flujo para la supresión de llaves de un árbol-B.

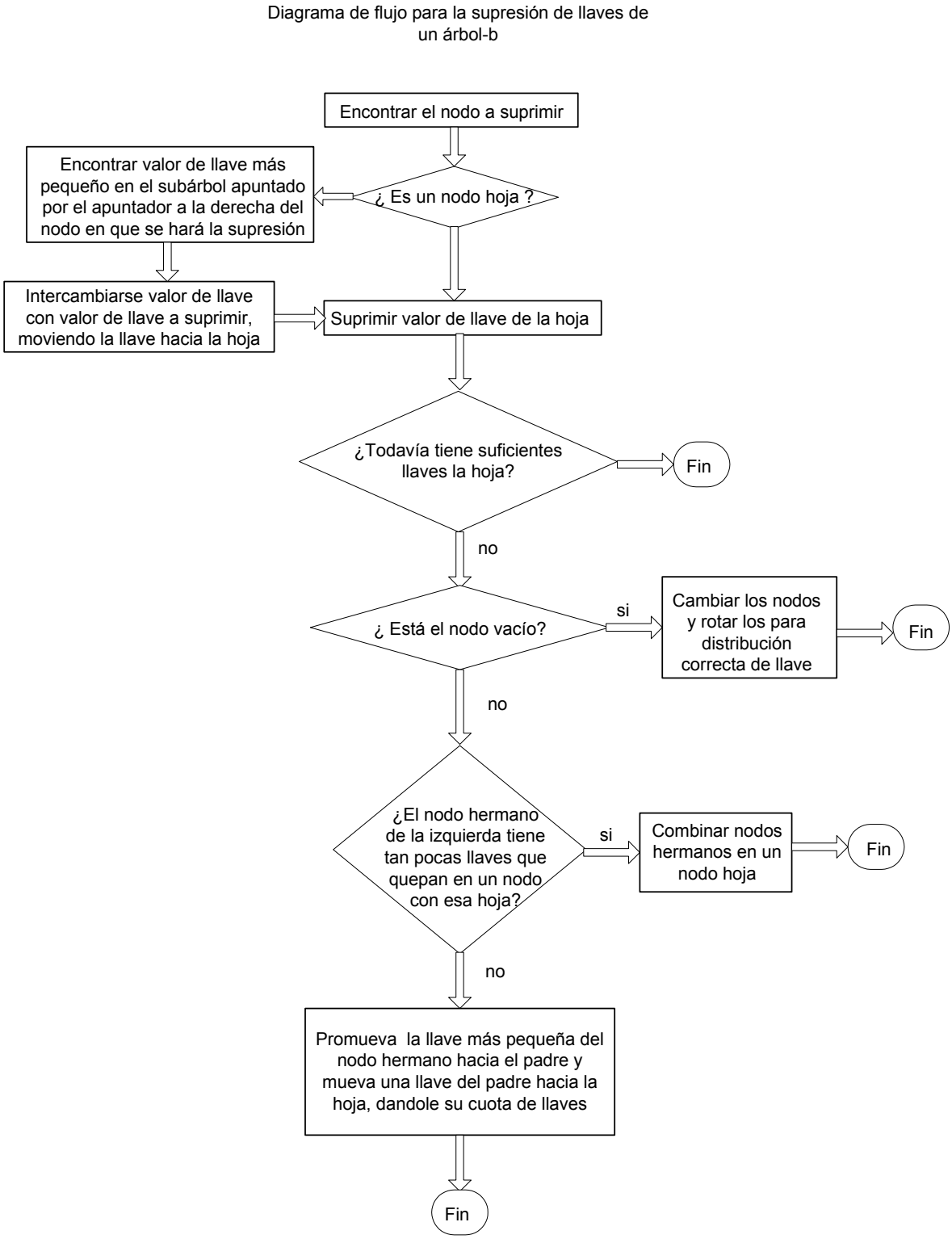
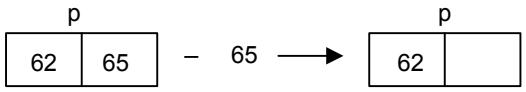


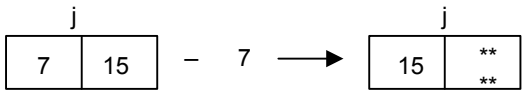
Ilustración 6-63: Diagrama de flujo para la supresión de llaves en un árbol B

Ejemplo 1:
Considere la supresión de la siguiente secuencia de llaves del árbol de la Ilustración 6-62.

Remover la llave con **valor 65** del **nodo p** es fácil.



De igual manera se puede remover la llave con **valor 7** del **nodo j**.



Note que la llave con valor 15 fue recorrida hacia la izquierda para mantener la secuencia apropiada de llaves.

Remover la llave con valor 40 deja al nodo e' con dos apuntadores pero sin llaves. Alguna llave debe ser localizada para guardarla en el nodo e', de tal forma que el requerimiento mínimo de llave quede satisfecho. Un valor de llave adecuado es la llave más a la izquierda del subárbol apuntado por el apuntador, después de la supresión de la llave; aquí el valor de la llave es 41 en el nodo n. La Ilustración 6-64, muestra lo expuesto.

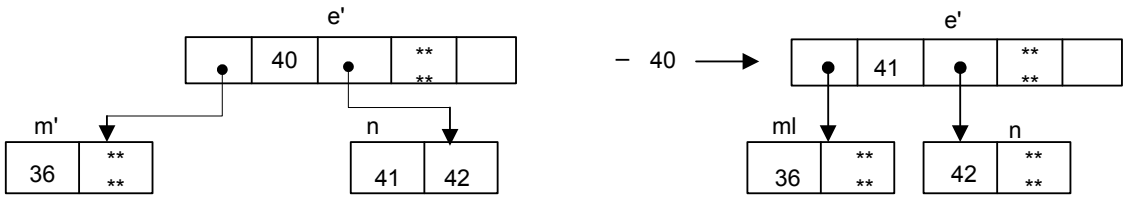


Ilustración 6-64: Removiendo la llave 40

Algunas veces las llaves están distribuidas de tal forma que los nodos deben ser combinados para conservar las restricciones del árbol-B. La supresión de la llave con valor 16 del nodo d, nos produce este caso. No sólo d y e son combinados, sino que j y k también lo son y también es afectado el nodo ancestro b. La Ilustración 6-65, muestra lo expuesto.

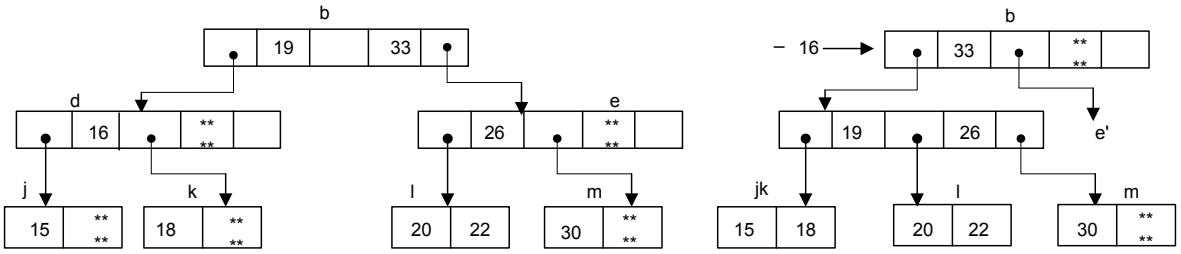


Ilustración 6-65: Removiendo la llave 16

Ejemplo 2:
En los ejemplos siguientes se muestra el estado del arbol antes y después de hacer la eliminación.

La Ilustración 6-66, muestra la eliminación de la clave 21 del árbol de la izquierda.

ELIMINACION: CLAVE 21

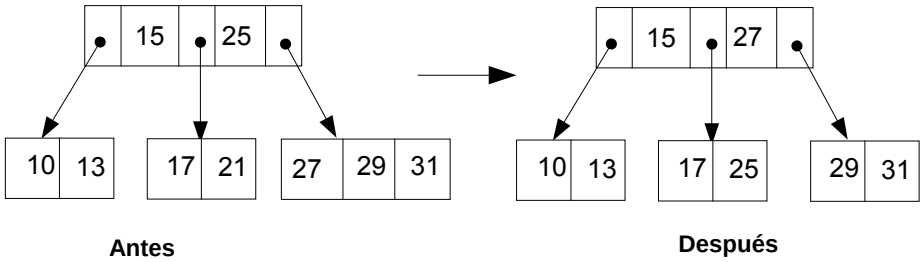


Ilustración 6-66: Ejemplo 2 de supresión de claves: eliminando la clave 21

La Ilustración 6-67, muestra la eliminación de la clave 10 del árbol de la izquierda.

ELIMINACION: CLAVE 10

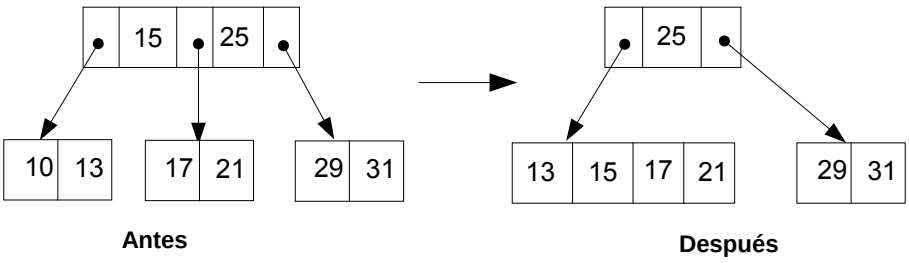


Ilustración 6-67: Ejemplo 2 de supresión de claves: eliminando la clave 10

La Ilustración 6-68, presenta la eliminación de la clave 15 del árbol de la izquierda.

ELIMINACION: CLAVE 15

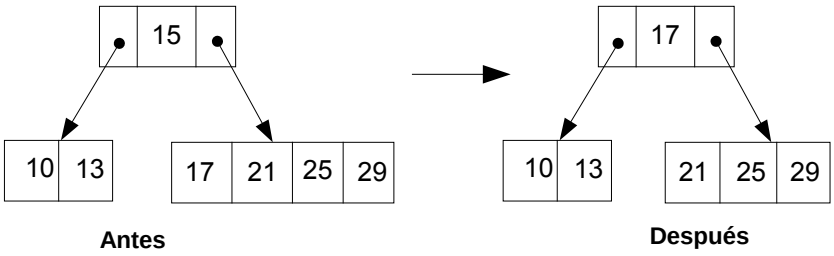


Ilustración 6-68: Ejemplo 2 de supresión de claves: eliminando la clave 15

La Ilustración 6-69, muestra la eliminación de la clave 25 del árbol de la izquierda.

ELIMINACION: CLAVE 25

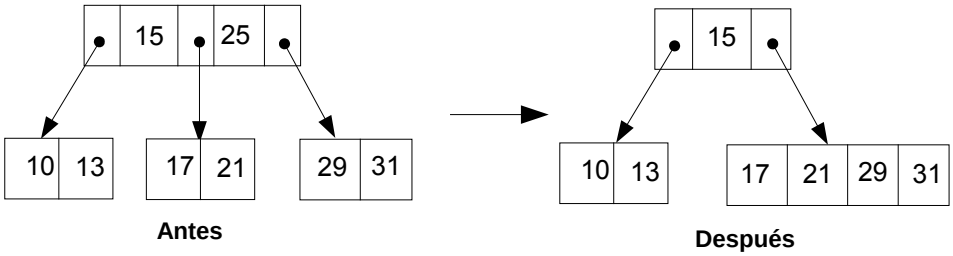


Ilustración 6-69: Ejemplo 2 de supresión de claves: eliminando la clave 25

6.4.5 Árboles-B+

Una de las técnicas populares para instrumentar la organización secuencial indexada de archivos es el uso de una variante de árbol-B básico, conocida como árbol-B+.

La estructura de índice de árbol-B+, es la más extendida de las estructuras de índices que mantienen su eficiencia a pesar de la inserción y borrado de datos ya que esto implica una degradación del rendimiento, además de un espacio extra.

Este tiempo adicional es aceptable incluso en archivos de altas frecuencias de modificación, ya que se evita el coste de reorganizar el archivo. Un índice de árbol B+ toma la forma de un árbol equilibrado donde los caminos de la raíz a cada hoja del árbol son de la misma longitud. Cada nodo que no es una hoja tiene [n / 2] y n hijos, donde n es fijo para cada árbol en particular.

Los árboles-B+ se han convertido en la técnica más utilizada para la organización de archivos indizados. La principal característica de estos árboles es que todas las claves se encuentran en las hojas y por lo tanto, cualquier camino desde la raíz hasta alguna de las claves, tienen la misma longitud.

Es de notar que los árboles-B+ ocupan un poco más de espacio que los árboles-B, y esto ocurre al existir duplicidad en algunas claves. Sin embargo, esto es aceptable si el archivo se modifica frecuentemente, puesto que se evita la operación de reorganización del árbol, que es tan costosa en los árboles-B.

Formalmente se define un árbol-B+ de la siguiente manera:

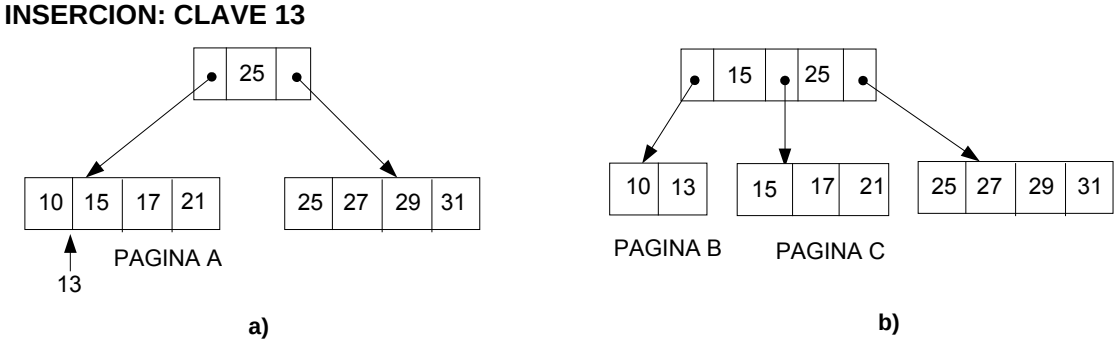
1. Cada página, excepto la raíz, contiene entre d y 2d elementos.
2. Cada página, excepto la raíz, tiene entre d + 1 y 2d + 1 descendientes. Se utiliza m para expresar el número de elementos por página.
3. La página raíz tiene al menos dos descendientes.
4. Las páginas hojas están todas al mismo nivel.
5. Todas las claves se encuentran en las páginas hojas.
6. Las claves de las páginas raíz e interiores se utilizan como índices.

Búsqueda de Árboles-B+

La operación de búsqueda en árboles-B+ es similar a la operación de búsqueda en árboles-B. El proceso es simple, sin embargo puede suceder que al buscar una determinada clave, la misma se encuentra en una página raíz o interior; en dicho caso no debe detenerse el proceso, sino que debe continuarse la búsqueda con la página apuntada por la rama derecha de dicha clave.

Inserción en árboles B+

El proceso de inserción en árboles-B+ es relativamente simple, similar al proceso de inserción en árboles-B. La dificultad se presenta cuando desea insertarse una clave en una página que se encuentra llena (m = 2d). En este caso, la página afectada se divide en 2, distribuyéndose las m + 1 claves de la siguiente forma: " las d primeras claves en la página de la izquierda y las d + 1 restantes claves en la página derecha". Una copia de la clave del medio sube a la página antecesora. En la Ilustración 6-70, se muestran dos diagramas del funcionamiento de este caso.

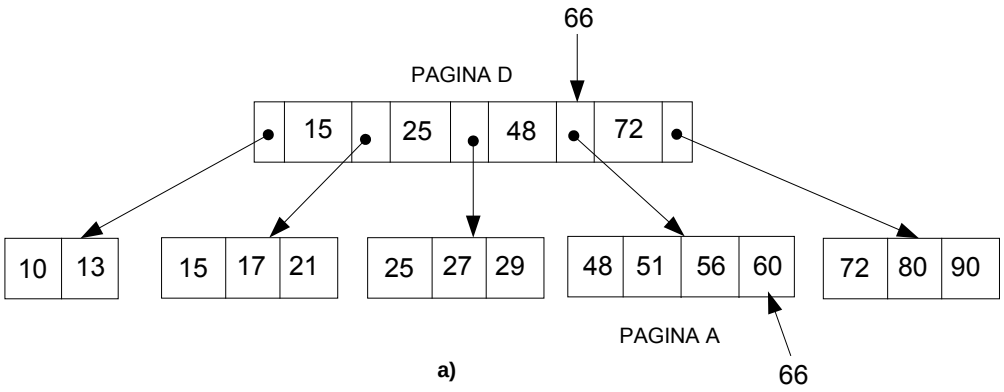


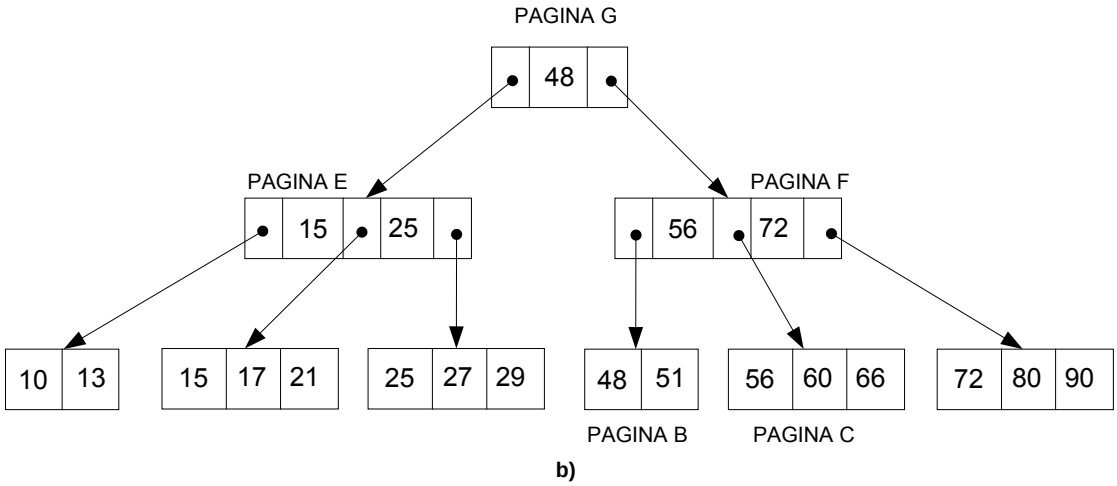
a)Antes de insertar la clave. b)Depués de insertarla.

Ilustración 6-70: Inserción de la clave 13 en un árbol B+

Puede suceder que la página antecesora se desborde nuevamente, entonces tendrá que repetirse el proceso anterior. Es importante notar que el desbordamiento en una página que no es hoja no produce duplicidad de claves. El proceso de propagación puede llegar hasta la raíz, en cuyo caso la altura del árbol puede incrementarse en una unidad. En la Ilustración 6-71, se presentan dos diagramas que clarifican y resuelven este caso.

INSERCION: CLAVE 66





a)Antes de insertar la clave. b)Depués de insertarla.

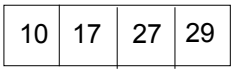
Ilustración 6-71: Inserción de la clave 66 en un árbol B+

Ejemplo

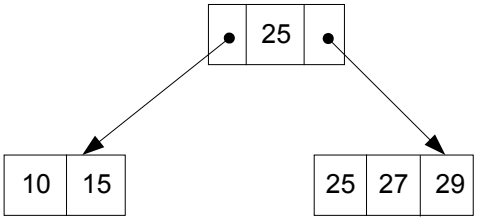
Supóngase que se desea insertar las siguientes claves en un árbol-B+ de **orden 2** que se encuentra vacío:
claves: 10-27-29-17-25-21-15-31-13-51-20-24-48-19-60-35-66

Los resultados parciales que ilustran el crecimiento del árbol se presentan en los siguientes diagramas correspondientes a la Ilustración 6-72.

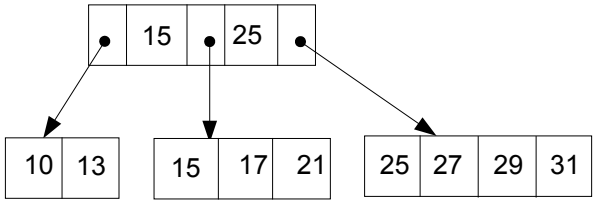
INSERCION: CLAVES 10, 27, 29 y 17



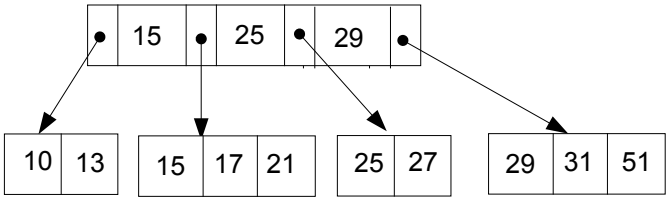
INSERCION: CLAVE 25



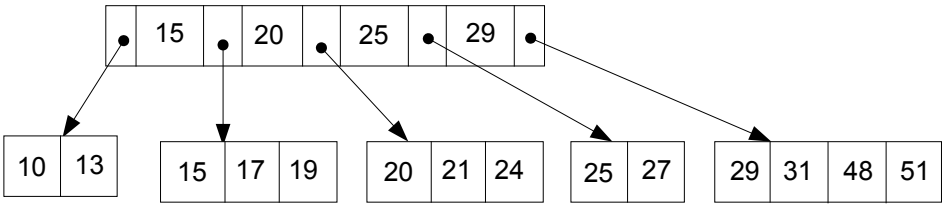
INSERCION: CLAVE 13



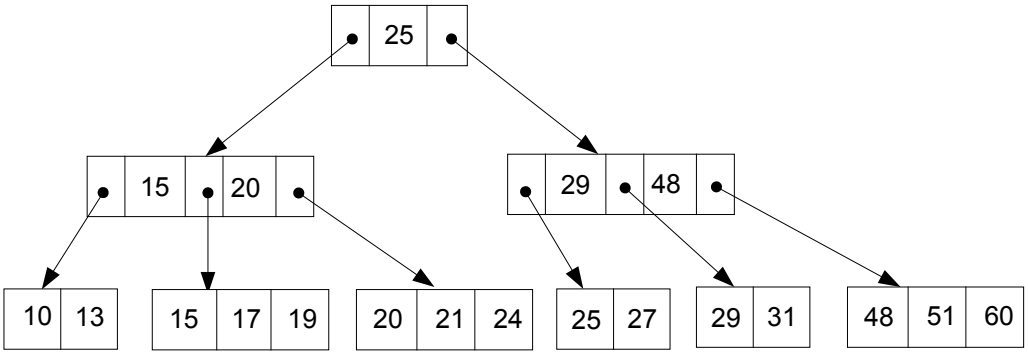
INSERCION: CLAVE 51



INSERCION: CLAVES 20, 24, 48 y 19



INSERCION: CLAVE 60



INSERCION: CLAVES 35 y 66

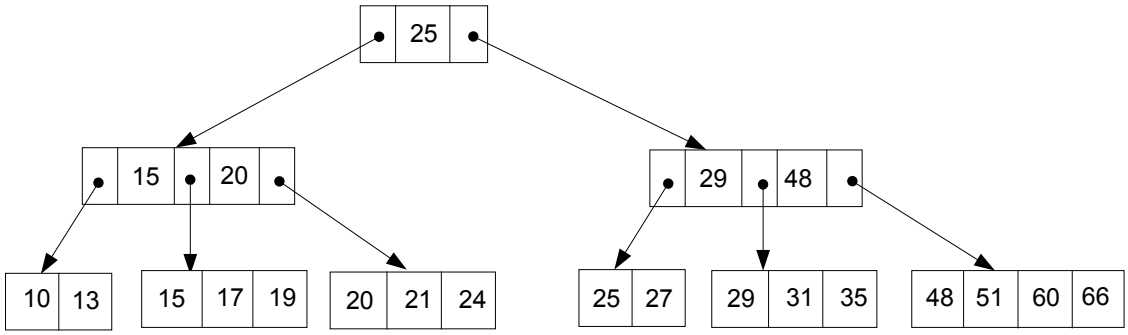


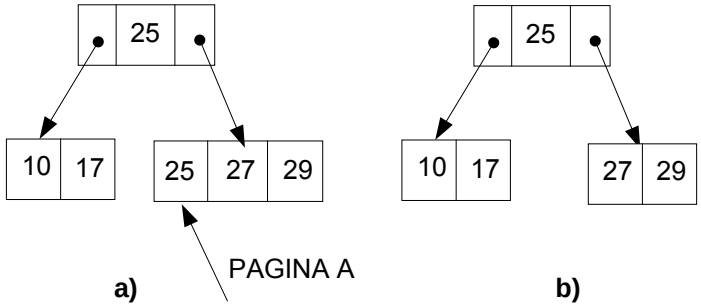
Ilustración 6-72: Inserción de claves en un árbol B+

Borrado en Árboles-B+

La operación de borrado en árboles-B+ es mas simple que la operación de borrado en árboles-B. Esto ocurre porque las claves a eliminar siempre se encuentran en las páginas hojas. En general deben distinguirse los siguientes casos:

1. Si al eliminar una clave, m queda mayor o igual a d entonces termina la operación de borrado. Las claves de las páginas raíz o internas no se modifican por más que sean una copia de la clave eliminada en las hojas. (Se presenta un ejemplo de este caso en la Ilustración 6-73).

ELIMINACIÓN: CLAVE 25

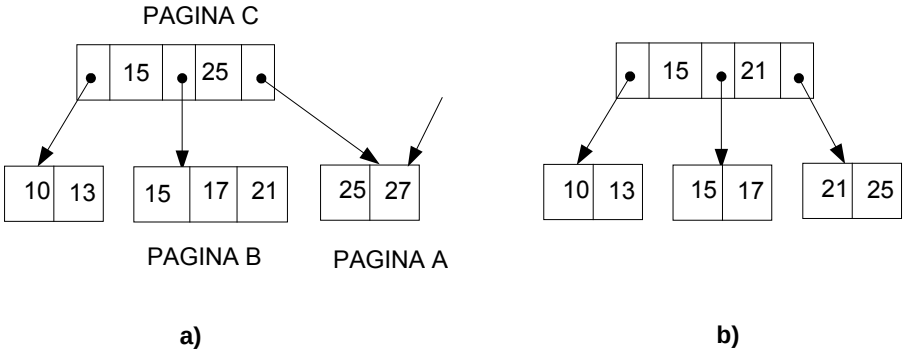


a)Antes de Eliminarla. b)Depués de Eliminarla.

Ilustración 6-73: Eliminación de una clave en un árbol B+

2. Si al eliminar una clave, m queda menor a d, entonces debe realizarse una redistribución de claves, tanto en el índice como en las páginas hojas. La Ilustración 6-74, muestra un primer ejemplo de cómo funciona esta técnica.

ELIMINACION: CLAVE 27

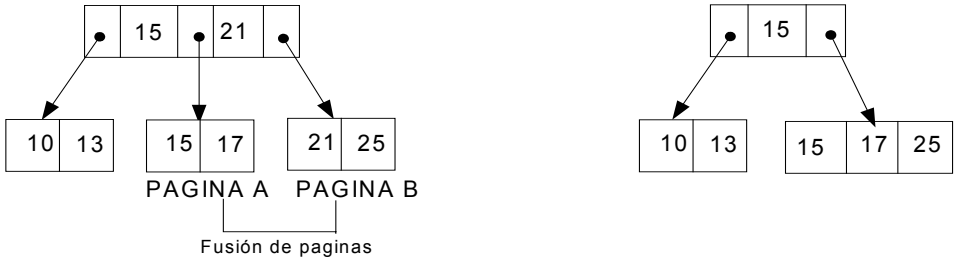


a)Antes de Eliminarla. b)Depués de Eliminarla.

Ilustración 6-74: Primer ejemplo de Redistribución de claves en un árbol B+

Nota: Al eliminar la clave 27 de la página A, m queda menor a d por lo que debe realizarse una redistribución de las claves. Se toma la clave que se encuentra más a la derecha en la rama izquierda de 25 (21 de la página B). Se coloca dicha clave en la página A y una copia de la misma, como índice, en la página C. La Ilustración 6-75, muestra un segundo ejemplo de cómo funciona esta técnica.

ELIMINACION: CLAVE 21



a)Antes de Eliminarla. b)Depués de Eliminarla.

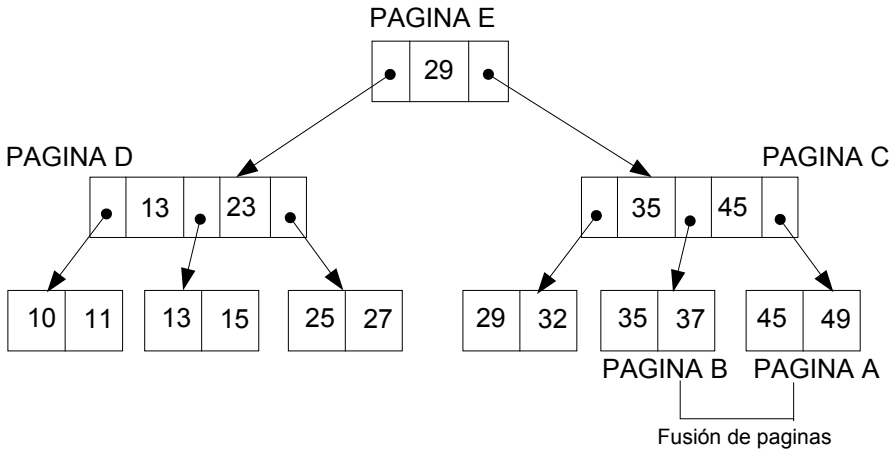
Ilustración 6-75: Segundo ejemplo de Redistribución de claves en un árbol B+

Nota: Al eliminar la clave 21 de la página A, m queda menor a d por lo que debe realizarse una redistribución de claves. Como no se puede tomar una clave de la página B puesto que m quedaría menor a d, entonces se realiza una fusión de las páginas A y B.

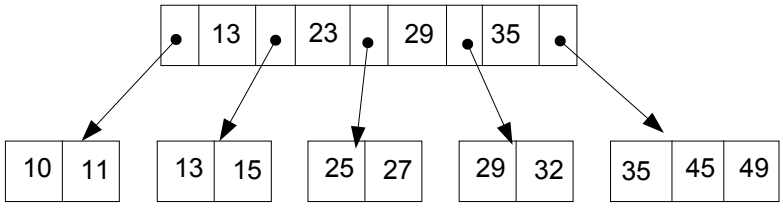
Puede suceder que al eliminar una clave y al realizar una redistribución de las mismas, la altura del árbol disminuya en una unidad. En la Ilustración 6-76 se presentan dos diagramas que clarifican y resuelven este caso.

ELIMINACION: CLAVE 37

a)



b)



a)Antes de Eliminarla. b)Depués de Eliminarla.

Ilustración 6-76: Disminución de la altura del árbol al realizar una distribución de claves

Nota: Al eliminar la clave 37 de la página A, m queda menor a d por lo que debe realizarse una redistribución de claves. Como no puede tomarse una clave de la página B puesto que m quedaría menor a d, entonces se realiza una fusión de las páginas A y B. Sin embargo, luego de está fusión m queda menor a d en la página C, por lo que debe bajarse la clave 29 de la página E y realizarse una nueva fusión, ahora de las páginas C y E. La altura del árbol disminuye en una unidad.

Ejemplo 1:

Suponga que desea eliminar las siguientes claves del árbol-B+ de orden 5 de la Ilustración 6-77:

Claves a eliminar: 15-51-48-60-31-20-10-25-17-24

Eliminar las claves:15-51-48-60-31-20-10-25-17-24

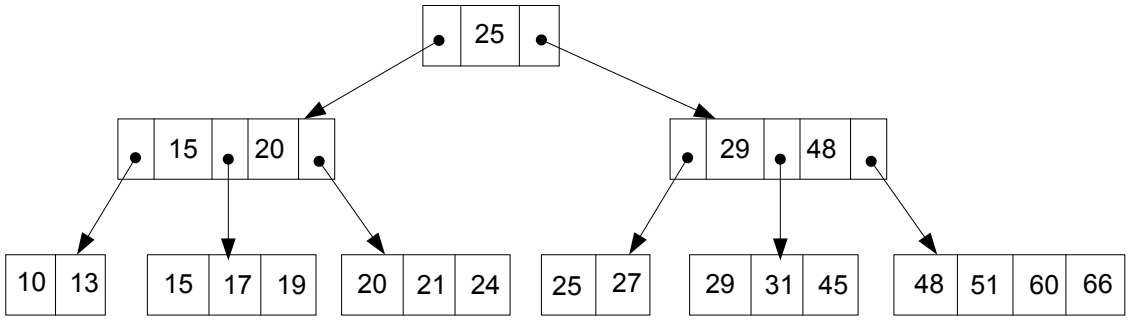


Ilustración 6-77: Árbol B+ para ejemplificar la eliminación de claves

La Ilustración 6-78, muestra el proceso de eliminar las claves 15, 51 y 48.

Eliminación de las claves:15-51-48

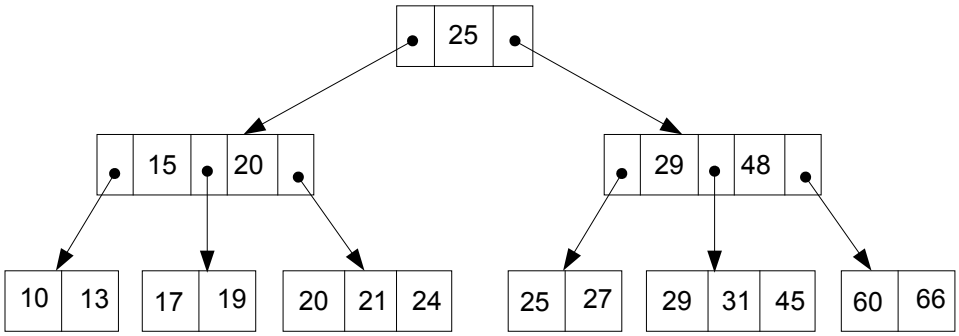


Ilustración 6-78: Eliminación de las claves 15, 51 y 48 en un árbol B+

La Ilustración 6-79, muestra el proceso de eliminar la clave 60.

Eliminación de la clave:60

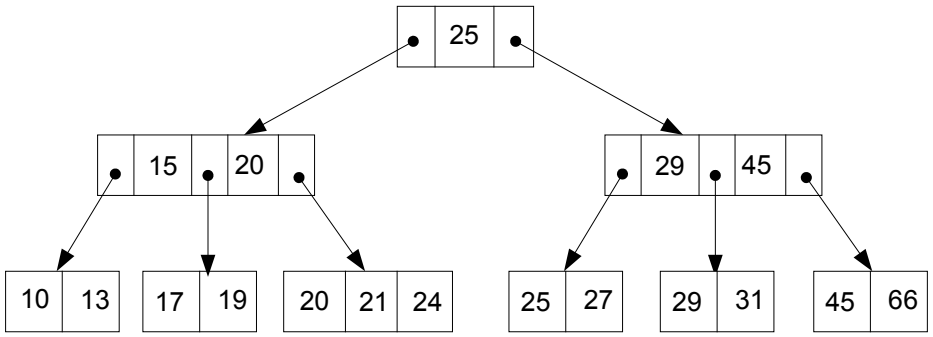


Ilustración 6-79: Eliminación de la clave 60 en un árbol B+

La Ilustración 6-80, presenta el proceso de eliminar la clave 31.

Eliminación de la clave:31

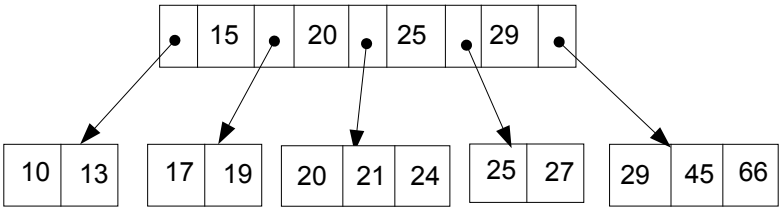


Ilustración 6-80: Eliminación de la clave 31 en un árbol B+

La Ilustración 6-81, muestra el proceso de eliminar las claves 20 y 26.

Eliminación de las claves: 20 y 66

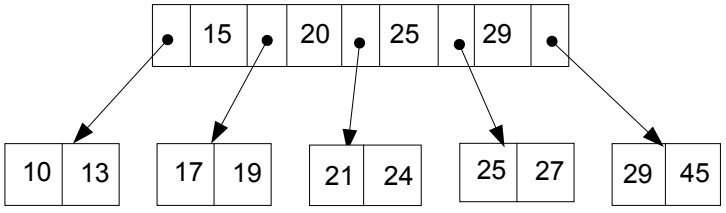


Ilustración 6-81: Eliminación de las claves 20 y 26 en un árbol B+

La Ilustración 6-82, muestra el proceso de eliminar las claves 29 y 10.

Eliminación de las claves: 29 y 10

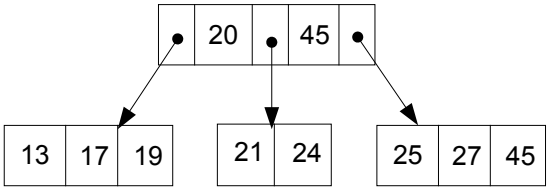


Ilustración 6-82: Eliminación de las claves 29 y 10 en un árbol B+

La Ilustración 6-83, muestra el proceso de eliminar las claves 25, 17 y 24.

Eliminación de las claves: 25-17 y 24

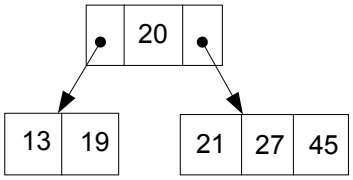


Ilustración 6-83: Eliminación de las claves 25, 17 y 24 en un árbol B+

Ejemplo 2:

Dibuje un árbol-B+ que corresponda al siguiente conjunto de valores de llaves:69, 20, 43, 110, 136, 15, 18, 30, 40, 58, 62, 100, 128, 130, 140, 7, 16, 19, 26, 36, 42, 50, 60, 65, 70, 120, 132, 138, 145, 150.

La solución del árbol se muestra en la Ilustración 6-84.

Los nodos de ‘a’ hasta ‘i’ forman parte del índice; los nodos de ‘j’ hasta ‘y’ forman el conjunto de secuencia. Las hojas han sido conectadas para formar una lista ligada de las llaves, en orden secuencial. Las llaves pueden ser accesadas eficientemente tanto directa como secuencialmente.

El acceso secuencial a las llaves de un árbol B no proporciona un buen rendimiento, especialmente cuando los requerimientos de memoria, para seguir las trayectorias a través del árbol, son considerados. El conjunto de secuencia de un árbol_B+ soluciona el problema.

.

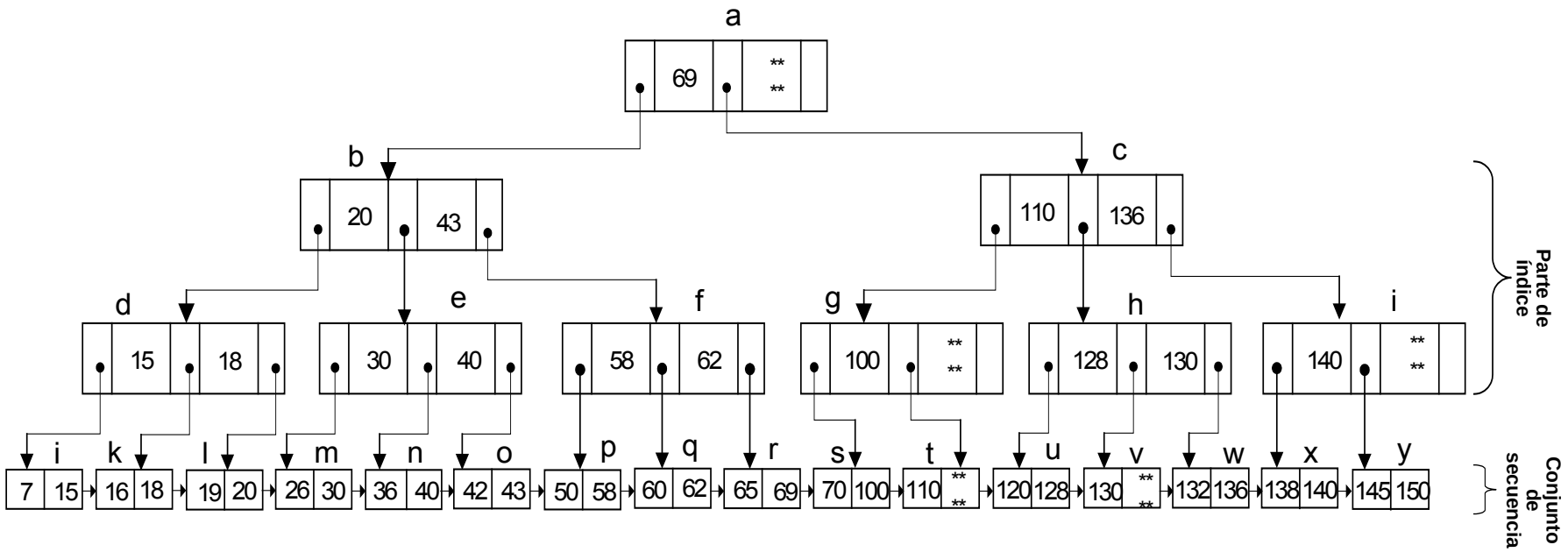


Ilustración 6-84: Árbol B+ resultante del ejemplo 2

Cuando se utiliza un árbol_B+ para proporcionar una organización secuencial indexada del archivo, los valores de llave en la parte de índices sólo existen para propósitos internos de dirigir el acceso al conjunto de secuencias

EJEMPLOS SENCILLOS DE ÁRBOLES EN LENGUAJE C

EJEMPLO 1

El siguiente ejemplo construye un árbol B, que maneja información relacionada con datos de estudiantes. Se usará la siguiente estructura de datos, para representar los nodos de un árbol:

```
typedef struct
{
        int  matricula;
        char nombre[30];
        char carrera[50];
}Reg_Alumno;

typedef struct datos nodo;          // tipo de nodo
struct datos                        // estructura de un nodo del arbol
{
        Reg_Alumno reg;
        int contador;
        nodo *izdo;                 // puntero a la raiz del subarbol izquierdo
        nodo *dcho;                 // puntero a la raiz del subarbol derecho
};
```

La idea en general será, partir de un árbol vacío y, a medida que se van insertando nuevos alumnos, el árbol va siendo construido respetando las reglas de construcción de un árbol B. Al final, se muestra el árbol construido, realizando para ello un recorrido en **IN-ORDEN**.

Antes de finalizar el programa, se escribe el contenido del árbol al disco.

El código del programa se muestra a continuación. De nuevo, el programa se ha dividido en un archivo de cabecera y en un archivo principal de implementación.

```
/* globales.h */
/* globales.h */

/* Estructuras de datos y funciones */

typedef struct
{
        int  matricula;
        char nombre[30];
        char carrera[50];
}Reg_Alumno;

typedef struct datos nodo;          // tipo de nodo
```

```
struct datos                                    // estructura de un nodo del arbol
{
        Reg_Alumno reg;
        int contador;
        nodo *izdo;                             // puntero a la raiz del subarbol
izquierdo
        nodo *dcho;                             // puntero a la raiz del subarbol derecho

};

int tam_nodo = sizeof (nodo);   // tamano de la estructura datos

// Declaracion de las funciones prototipos
void error();
void buscar(Reg_Alumno, nodo **);
void visualizar_arbol(nodo *, int);
int LeerDatosAlumno (Reg_Alumno *);
void GrabarArbolEnDisco (nodo *, FILE *);


// Funcion Error()
void error(void)
{
        perror("eror: Insuficiente espacio de memoria");
        exit(1);

}

nodo *NuevoNodo()
{
        nodo *q = (nodo *) malloc (sizeof(nodo));
        if (!q)
                error();

        return (q);

}

nodo *raiz = NULL;              // Apunta a la raiz del árbol

/*************************************************************************
                        Buscar una clave en el árbol
  *************************************************************************/

/* Buscar por un determinado nodo y si no esta insertarlo
   el valor para a es pasado por referencia para hacer posibles
   los enlaces entre nodo y nodo cuando se crea uno de estos*/

#define ArbolVacio (a == NULL)

void buscar(Reg_Alumno registro, nodo **raiz)
```

```
{
        nodo *a;
        a = *raiz;      // Raíz del árbol

        if (ArbolVacio)// El nodo con clave matricula, no está en el árbol
        {       // Insertarlo
                a = NuevoNodo();
                a->reg = registro;
                a->contador = 1;
                a->izdo = a->dcho = NULL;
        }

        else    // El árbol no está vacío
        {
                if (registro.matricula  < a->reg.matricula)
                        // El valor buscado está a la izquierda de este nodo
                        buscar (registro, &a->izdo);
                else
                {
                        if (registro.matricula > a->reg.matricula)
                                // El valor buscado está a la derecha de este nodo
                                buscar (registro, &a->dcho);

                        else    // El valor buscado existe
                                a->contador++;

                }
        }

        *raiz = a;
} // Fin de la función buscar ()

/*********************************************************************
                Visualizar el árbol
  ********************************************************************/

/* Visualizar el árbol a con margen n
   se emplea la forma inorden, para recorrer el árbol*/

void visualizar_arbol (nodo *a, int n)
{

        int i;

        if (!ArbolVacio)
        {
                visualizar_arbol (a->izdo, n + 1);

                for (i = 1; i <= n; i++)
                        printf("     ");
```

```
                printf ("%d %s (%d)  \n", a->reg.matricula, a->reg.nombre, a->contador);
                visualizar_arbol (a->dcho, n + 1);
        }
} // fin de visualizar ()

// Leer datos de un alumno
int LeerDatosAlumno (Reg_Alumno *reg)
{
        int resp_matri;

        printf ("\n\t Matricula: ");
        resp_matri = scanf ("%d", &reg->matricula);

        if (resp_matri == EOF) // EOF equivalente a -1
                return (resp_matri);    /* Si se tecleó ^Z, entonces regresar */

        fflush (stdin);

        printf ("\t Nombre: ");
        gets (reg->nombre);

        printf ("\t Carrera: ");
        gets (reg->carrera);

        return (1);

} // Fin de LeerDatosAlumno


/* Grabar el árbol en el disco, recorriendolo en inorden:
   primero se visita el nodo izquierdo, luego la raiz, y por último
   el nodo derecho
*/
void GrabarArbolEnDisco (nodo *raiz, FILE *puntero_fich)
{

        if (raiz != NULL)
        {
                GrabarArbolEnDisco (raiz->izdo, puntero_fich);

                fwrite (raiz, tam_nodo, 1, puntero_fich);

                GrabarArbolEnDisco (raiz->dcho, puntero_fich);
        }

}
```

/* principal.cpp */

```
/* principal.cpp */

#include <stdio.h>
#include <stdlib.h>
#include "globales.h"

void main()
{
        FILE *pf;
        int i = 1;
        int resp;
        Reg_Alumno registro;          // datos de un alumno
        nodo Nodo;

        system("cls");

        pf = fopen("arbol", "rb");  // abrir el fichero arbol para leer

        if (pf != NULL)
        {
                // El fichero existe, reconstruir la lista en MP
                // Traer datos del disco
                printf ("\n\t Trayendo datos del disco....");
                fread (&Nodo, tam_nodo, 1, pf);

                while (!feof (pf) && !ferror (pf))
                {
                        buscar (Nodo.reg, &raiz);      // raiz se pasa por referencia
                        fread (&Nodo, tam_nodo, 1, pf);
                }
                printf ("\n\t Datos traidos del disco....\n\n");

                fclose (pf);

                printf ("\n\t El arbol traido del disco es..... \n\n");
                visualizar_arbol (raiz, 0);

        }

        printf ("\n\n\n\t Introduzca datos de estudiantes \n");
        printf ("\n\t Finalizar tecleando matricula con ^Z \n\n");

        printf ("\n\t DATOS DEL ALUMNO %d", i);
        resp = LeerDatosAlumno (&registro);

        while (resp != EOF)
        {
                buscar (registro, &raiz);        // raiz se pasa por referencia
                i++;
```

```
            printf ("\n\n\t DATOS DEL ALUMNO %d", i);
            resp = LeerDatosAlumno (&registro);
        }

        // Visualizar el árbol
        printf ("\n\t El arbol recorrido en in-orden es: \n");
        visualizar_arbol (raiz, 0);

        printf ("\n\t Iniciada la copia de datos al disco...");

        pf = fopen("arbol", "wb"); // Abrir el fichero arbol para escribir
        GrabarArbolEnDisco (raiz, pf);
        printf ("\n\t Datos copiados al disco \n\n");
        fclose (pf);

} // fin del main()
```

EJEMPLO 2

Un árbol binario está perfectamente equilibrado si, para todo nodo, el número de nodos en el subárbol izquierdo y el número de nodos en el subárbol derecho, difieren como mucho en una unidad. Como ejemplo, vamos a considerar el problema de construir un árbol perfectamente equilibrado, siendo los valores de los nodos, **n** números que se leen de un fichero de datos, en nuestro caso del fichero predefinido stidn (fichero estándar de entrada).
Esto puede realizarse fácilmente distribuyendo los nodos, según se leen, equitatativamente a la izquierda y a la derecha de cada nodo. El proceso recursivo que se indica a continuación, es la mejor forma de realizar esta distribución. Para un número dado de **n** nodos, y siendo **ni** (nodos a la izquierda) y **nd** (nodos a la derecha) dos enteros, el proceso es el siguiente:

1. Utilizar un nodo para la raíz.

2. Generar el subárbol izquierdo con **ni = n / 2** nodos utilizando la misma regla.

3. Generar el subárbol derecho con **nd = n – ni – 1** nodos utilizando la misma regla.

Cada nodo del árbol consta de los siguientes miembros: clave, puntero al subárbol izquierdo y puntero al subárbol derecho.

Una propuesta de programa de la técnica descrita se presenta a continuación, en la cual se ha utilizado una función recursiva llamada **construir_arbol ( )**, la cual construye un árbol de **n** nodos.

De nuevo, el programa se ha dividido en 2 ficheros: fichero de cabecera con la declaración de tipos y definición de funciones, y un fichero principal de implementación.

/* globales.h */

```
/* globales.h */

typedef struct datos nodo;              // Tipo nodo
struct datos                                    // Estructura de un nodo del árbol
{
        int clave;
        nodo *izdo;
        nodo *dcho;
};

int tam_nodo = sizeof (nodo);   // tamano de la estructura datos

// Declaración de funciones prototipos
void error();
void visualizar_arbol (nodo *, int);
nodo *Construir_arbol (int);
nodo *Construir_arbol2 (int, FILE *);
void GrabarArbolEnDisco (nodo *, FILE *);


// Definición de funciones
void error (void)
{
        perror ("Error: insuficiente espacio de memoria");
        exit(1);
}


nodo *NuevoNodo()
{
        nodo *q = (nodo *) malloc (sizeof(nodo));

        if(!q)
                error();

        return(q);
}

nodo *raiz;                             // Apunta a raiz del árbol


/*************************************************************************
                                                Función construir árbol
  *************************************************************************/

/* construir un árbol de n nodos perfectamente equilibrados*/
nodo *Construir_arbol(int n)
{
        nodo *q;
        int ni, nd;
```

```
        if(n == 0)
                return(NULL);

        else
        {
                ni = n / 2;                     // Nodos del subárbol izquierdo
                nd = n - ni - 1; // Nodos del subárbol derecho
                q = NuevoNodo();
                printf("clave: ");
                scanf("%d", &q->clave);
                q->izdo = Construir_arbol(ni);
                q->dcho = Construir_arbol(nd);
                return (q);
        }
}


/* construir un árbol de n nodos perfectamente equilibrados*/
nodo *Construir_arbol2(int n, FILE *pf)
{
        nodo *q;
        int ni, nd;

        if (n == 0)
                return (NULL);

        else
        {
                ni = n / 2;                     // Nodos del subárbol izquierdo
                nd = n - ni - 1; // Nodos del subárbol derecho
                q = NuevoNodo();      // Creamos el nuevo nodo

                fread (q, tam_nodo, 1, pf);

                q->izdo = Construir_arbol2 (ni, pf);
                q->dcho = Construir_arbol2 (nd, pf);
                return (q);
        }
}

/************************************************************************
                                        Visualizar el árbol
  ************************************************************************/
/*              Visualizar el árbol a con márgen n
                Se emplea la forma inorden, para recorrer el árbol*/

void visualizar_arbol(nodo *a, int n)
{
        int i;
```

```
        if(a != NULL)           // Si el árbol no está vacío
        {
                visualizar_arbol(a->izdo, n + 1);

                for(i = 1; i <= n; i++)
                        printf("     ");

                printf("%d\n ", a->clave);
                visualizar_arbol(a->dcho, n + 1);

        }
}

/* Grabar el árbol en el disco, recorriendolo en inorden:
   primero se visita el nodo izquierdo, luego la raiz, y por último
   el nodo derecho
*/
void GrabarArbolEnDisco (nodo *raiz, FILE *puntero_fich)
{

        if (raiz != NULL)
        {
                GrabarArbolEnDisco (raiz->izdo, puntero_fich);

                fwrite (raiz, tam_nodo, 1, puntero_fich);

                GrabarArbolEnDisco (raiz->dcho, puntero_fich);
        }

}
```

/* principal.cpp */

```
/* principal.cpp */
// Árbol Perfectamente equilibrado

#include <stdio.h>
#include <stdlib.h>
#include "globales.h"

void main()
{
        int n;


        FILE *pf, *pf1;
        system("cls");

        pf = fopen("arbol", "rb");  // abrir el fichero arbol para leer
        pf1 = fopen("cantidad", "rb");

        if (pf != NULL)
```

```
        {
                // El fichero existe, reconstruir la lista en MP
                // Traer datos del disco
                printf ("\n\t Trayendo datos del disco....");
                n = getw (pf1);
                raiz = Construir_arbol2(n, pf);          // Construir ärbol de n nodos

                printf ("\n\t El arbol traido del disco es..... \n\n");
                visualizar_arbol (raiz, 0);

                fclose (pf);
                fclose (pf1);

                exit (0);
        }

        printf("Numero de nodos:");
        scanf("%d", &n);

        printf("Introducir claves:\n \n");
        raiz = Construir_arbol(n);      // Construir ärbol de n nodos

        visualizar_arbol(raiz, 0);


        printf ("\n\t Iniciada la copia de datos al disco...");

        pf = fopen("arbol", "wb"); // Abrir el fichero arbol para escribir
        GrabarArbolEnDisco (raiz, pf);
        printf ("\n\t Datos copiados al disco \n\n");
        fclose (pf);

        // Escribimos en un fichero aparte la cantidad de nodos
        pf1 = fopen("cantidad", "wb");
        putw(n, pf1);
        fclose (pf1);

}
```

6.4.6 Funciones de cálculo de dirección

Técnica HASHING

Permite ahorrarse el recorrido de una estructura de índice. Se basa en el cálculo de la dirección de un dato, directamente calculando la función sobre el valor de la llave de búsqueda del registro deseado.

¿Cómo elegir una función para aplicar la técnica hashing?

- Buscar una distribución uniforme (el número de llaves por cubeta de datos igual para todas las cubetas).

- Idealmente todas las cubetas están compuestas por una sola página (evitar los punteros de una cubeta a otra cuando se sobrepasan el número de llaves por cubetas).

Ejemplo de elección de una mala función hashing para trabajar con llaves que son cadenas de caracteres.

- Las tres primeras letras del valor string (es una función muy sencilla pero no resulta una distribución uniforme: van a almacenarse más elementos por ejemplo con A / M / N que con Z / U).

Una posible corrección del ejemplo anterior: La suma de todos los caracteres del string donde la letra número i del alfabeto (gringo) está representa por el entero (i) modulo b (donde b es el número de cubetas).

Utilizando la función módulo realice la tabla de cálculo de dirección para el archivo depósito empleando nombre_sucursal como llave, suponiendo un total de 10 cubetas (0…9).

Tabla 6-7: Tabla DEPÓSITO para el cálculo de dirección

Brigton	217	Green	750
Downtown	101	Jonson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Perryridge	201	Williams	900
Perryridge	218	Lyle	700
Redwood	222	Lindsay	700
Roun Hill	305	Turner	350

Mapa de letras

A(1)	B(2)	C(3)	D(4)	E(5)	F(6)	G(7)	H(8)	I(9)
J(10)	K(11)	L(12)	M(13)	N(14)	O(15)	P(16)	Q(17)	R(18)
S(19)	T(20)	U(21)	V(22)	W(23)	X(24)	Y(25)	Z(26)	

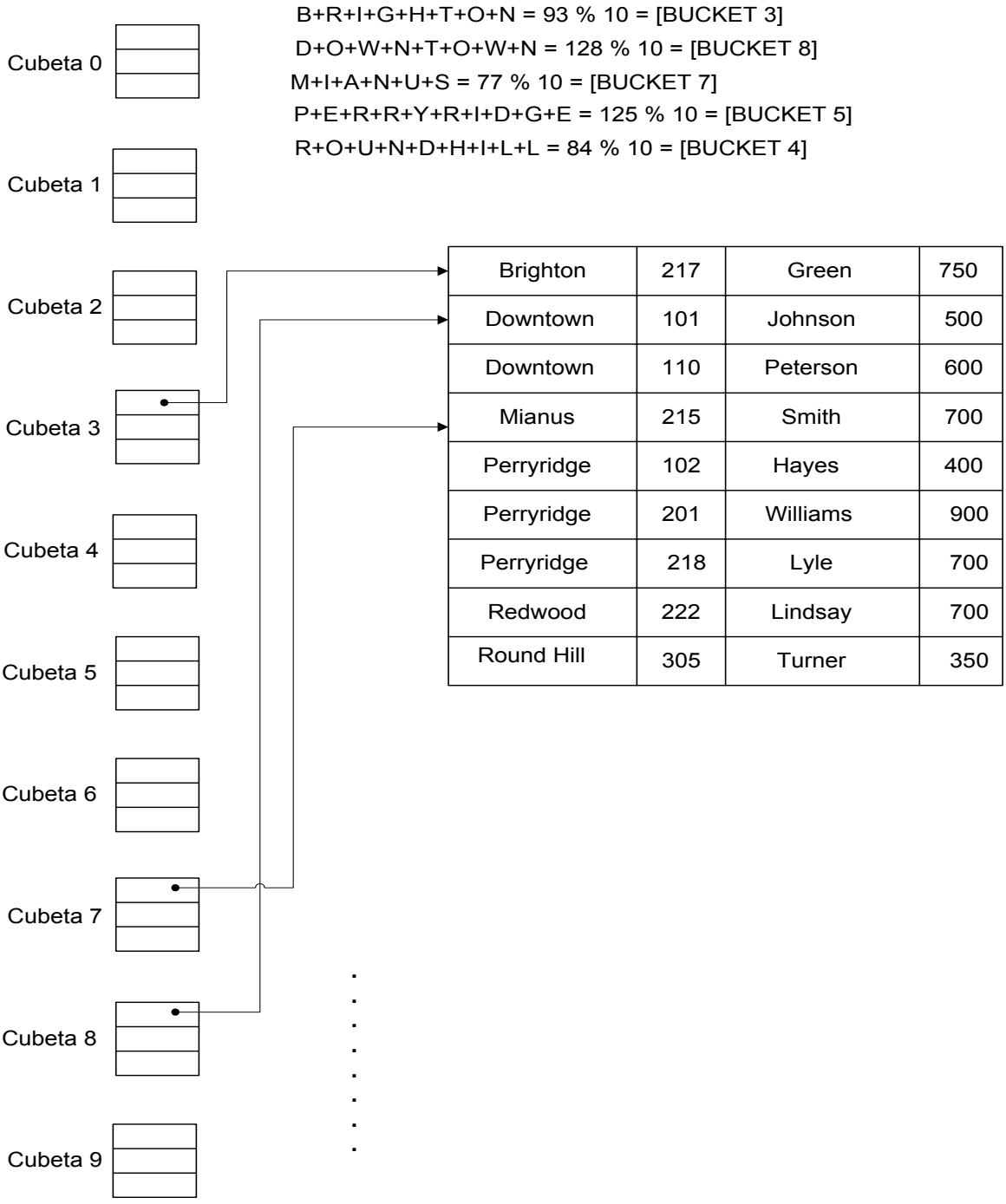


Ilustración 6-85: Tabla de cálculo de dirección para el ejemplo de la Tabla 6-7

CODIFICACIÓN EN LENGUAJE C DE LA TÉCNICA HASHING

A continuación se presenta una propuesta de programación en Lenguaje C, de la técnica descrita anteriormente.

Se ha usado la siguiente estructura,

```
typedef struct
{
        char carnet[20];        // Carnet del estudiante
        char nombre[50];        // Nombre del estudiante
        int edad;               // Edad del estudiante
}Info_Alumno;
```

para representar la información de un alumno.

Además, se ha usado la siguiente constante:

```
#define No_CUBETAS 10 // cantidad de cubetas
```

para representar un máximo de 10 cubetas.

La idea principal del programa, será pedir una cantidad de alumnos para ingresar sus datos. A continuación se pedirán los datos para cada alumno, y luego se imprimirá por cada nombre de alumno, el valor de sus letras y el peso del nombre completo. También se imprimirá, el estado de las cubetas, esto es, si tienen o no, nombres de alumnos asociados.

La codificación del programa se muestra a continuación: (El programa se ha dividido en 2 archivos: uno con extensión .h donde se han declarado las funciones y las estructuras de datos usadas, y el otro archivo con extensión .cpp donde se muestra el programa principal).

/* globales.h */

```
/* Archivo de cabecera para la implementación de las estructuras de datos
   y las definiciones de las funciones. */

typedef struct
{
        char carnet[20];        // Carnet del estudiante
        char nombre[50];        // Nombre del estudiante
        int edad;               // Edad del estudiante

}Info_Alumno;

#define No_CUBETAS 10       // cantidad de cubetas
```

/* Funciones prototipos */

```
void Error (void);
void LeerDatos (Info_Alumno *p, int no_al);
void Imprimir (Info_Alumno *p, int no_al);
int CalcularPeso (char *nom);
```

/* Definición de funciones */

```
/* Función Error */
void Error (void)
{
        perror ("Error: insuficiente espacio de memoria.\n");
        exit (1);
}

void LeerDatos (Info_Alumno *pAl, int no_al)
{
        int i;

        for (i = 0; i < no_al; i++)
        {
                printf ("\n\t DATOS DEL ALUMNO %d", i + 1);

                fflush (stdin);

                printf ("\n\t Carnet: ");
                gets (pAl[i].carnet);      // gets ((pAl + i)->carnet);

                printf ("\n\t Nombre: ");
                gets (pAl[i].nombre);

                printf ("\n\t Edad: ");
                scanf ("%d", &pAl[i].edad);

        }

}
```

```
void Imprimir (Info_Alumno *pAl, int no_al)
{
	int i;

	for (i = 0; i < no_al; i++)
	{
		printf ("\n\t ALUMNO %d ....", i + 1);
		printf ("\n\t Carnet: %s", pAl[i].carnet);
		printf ("\n\t Nombre: %s", pAl[i].nombre);
		printf ("\n\t Edad: %d", pAl[i].edad);
		printf ("\n\n");
	}

}

// Calcular el peso del nombre del alumno pasado
int CalcularPeso (char *nom)
{
	int let;
	int i = 0;
	int suma = 0;

	printf ("\n\t LETRA \t PESO");
	printf ("\n\t -------------------------");
	for (i = 0; i < (int) strlen (nom); i++)
	{
		let = toupper (nom[i]);		// Convertir a mayúscula, si procede
		let = let - 64;			// Calcular el valor en el mapa de letras
		suma = suma + let;		// Acumular la suma
		printf ("\n\t %c \t %d", toupper(nom[i]), let);
	}

	return (suma % No_CUBETAS);	// Obtener el módulo

}
```

/* principal.cpp */

```
/* Este programa implementa la técnica hashing simple. */

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <crtdbg.h>
#include "globales.h"   // Declaraciones y Definiciones de datos y funciones


void main()
{

	Info_Alumno *pIA = NULL;
```

```
Info_Alumno ***cubetas =  NULL;
int i = 0, no_alumnos = 0;
int peso_nombre;
int j = 0;
int ocurrencias[No_CUBETAS];

printf ("\n\t Cantidad de alumnos: ");
scanf ("%d", &no_alumnos);

pIA = (Info_Alumno *) malloc (no_alumnos * sizeof (Info_Alumno));

if (pIA == NULL)
        Error();

LeerDatos (pIA, no_alumnos);
Imprimir (pIA, no_alumnos);

// Asignar memoria para el array cubetas
cubetas = (Info_Alumno ***) malloc (No_CUBETAS * sizeof (Info_Alumno **));
if (cubetas == NULL)
        Error();

// Inicializar las cubetas
for (i = 0; i < No_CUBETAS; i++)
        cubetas[i] = NULL;

// Inicializar las ocurrencias de los pesos
for (i = 0; i < No_CUBETAS; i++)
        ocurrencias[i] = 0;

// Calcular los pesos del nombre de los alumnos
for (i = 0; i < no_alumnos; i++)
{
        peso_nombre = CalcularPeso (pIA[i].nombre);
        /* El valor del peso calculado está entre 0 y 9 */
        if (cubetas[peso_nombre] == NULL)
        {
                cubetas[peso_nombre] =  (Info_Alumno **) malloc (1 * sizeof
                                        (Info_Alumno *));
                cubetas[peso_nombre][0] = &pIA[i];
                ocurrencias[peso_nombre]++;
        }

        else    // Ya hay al menos un elemento apuntado
        {
                ocurrencias[peso_nombre]++;
                cubetas[peso_nombre] =  (Info_Alumno **) realloc
                                        (cubetas[peso_nombre],
                                        ocurrencias[peso_nombre] * sizeof
                                        (Info_Alumno *) );
                if (cubetas[peso_nombre] == NULL)
                        Error();
                j = ocurrencias[peso_nombre] - 1;
```

```
                cubetas[peso_nombre][j] = &pIA[i];
        }


        printf ("\n\t %s  --> Peso: %d", pIA[i].nombre, peso_nombre);
        printf ("\n");
}       // Fin del for


// Imprimir los índices (numeros de las cubetas) con sus respectivos nombres
printf ("\n\n\t ******** CUBETAS ASOCIADAS A NOMBRES ******* ");
for (i = 0; i < No_CUBETAS; i++)
{
        printf ("\n\t Cubeta %d -->: ", i);

        if (cubetas[i] == NULL)
                printf ("\n\t NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ...");

        else    // Al menos hay un nombre asociado
        {
                for (j = 0; j < ocurrencias[i]; j++)
                        printf (" %s ", cubetas[i][j]->nombre);
        }
        printf ("\n");

}       // Fin del for

// Liberar la memoria asignada a las cubetas
for (i = 0; i < No_CUBETAS; i++)
{
        if (cubetas[i] != NULL)
                free (cubetas[i]);

}

free (cubetas);

free (pIA);

if (_CrtDumpMemoryLeaks ())
        printf ("\n\t Hay Lagunas de memoria");

} /* Fin de main */
```

SALIDA DEL PROGRAMA

A continuación, se muestra una salida de la ejecución del programa. Como siempre, las entradas del usuario se muestran en negrita, cursiva y subrayado.

```
Cantidad de alumnos: 6

DATOS DEL ALUMNO 1
Carnet: c-01

Nombre: Juan

Edad: 21

DATOS DEL ALUMNO 2
Carnet: c-02

Nombre: Pedro

Edad: 20

DATOS DEL ALUMNO 3
Carnet: c-03

Nombre: Maria

Edad: 22

DATOS DEL ALUMNO 4
Carnet: c-04

Nombre: Juan

Edad: 24

DATOS DEL ALUMNO 5
Carnet: c-05

Nombre: Ana

Edad: 23

DATOS DEL ALUMNO 6
Carnet: c-06

Nombre: Pedro

Edad: 25


ALUMNO 1 ....
Carnet: c-01
Nombre: Juan
```

```
Edad: 21


ALUMNO 2 ....
Carnet: c-02
Nombre: Pedro
Edad: 20


ALUMNO 3 ....
Carnet: c-03
Nombre: Maria
Edad: 22


ALUMNO 4 ....
Carnet: c-04
Nombre: Juan
Edad: 24


ALUMNO 5 ....
Carnet: c-05
Nombre: Ana
Edad: 23


ALUMNO 6 ....
Carnet: c-06
Nombre: Pedro
Edad: 25


LETRA   PESO
--------------------------
J       10
U       21
A       1
N       14
Juan  --> Peso: 6

LETRA   PESO
--------------------------
P       16
E       5
D       4
R       18
O       15
Pedro  --> Peso: 8

LETRA   PESO
--------------------------
M       13
```

```
A       1
R       18
I       9
A       1
Maria  --> Peso: 2

LETRA   PESO
---------------------------
J       10
U       21
A       1
N       14
Juan  --> Peso: 6

LETRA   PESO
---------------------------
A       1
N       14
A       1
Ana  --> Peso: 6

LETRA   PESO
---------------------------
P       16
E       5
D       4
R       18
O       15
Pedro  --> Peso: 8


******** CUBETAS ASOCIADAS A NOMBRES *******
Cubeta 0 -->:
NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ...

Cubeta 1 -->:
NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ...

Cubeta 2 -->:  Maria

Cubeta 3 -->:
NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ...

Cubeta 4 -->:
NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ...
Cubeta 5 -->:
NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ...

Cubeta 6 -->:  Juan  Juan  Ana

Cubeta 7 -->:
NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ...
```

```
        Cubeta 8 -->:  Pedro  Pedro

        Cubeta 9 -->:
        NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ...
Press any key to continue
```

EXPLICACIÓN DE LA CODIFICACIÓN DEL PROGRAMA

En primer lugar, en la Ilustración 6-86, se muestran gráficamente las estructuras de datos y su relación entre ellas, usadas en el programa:

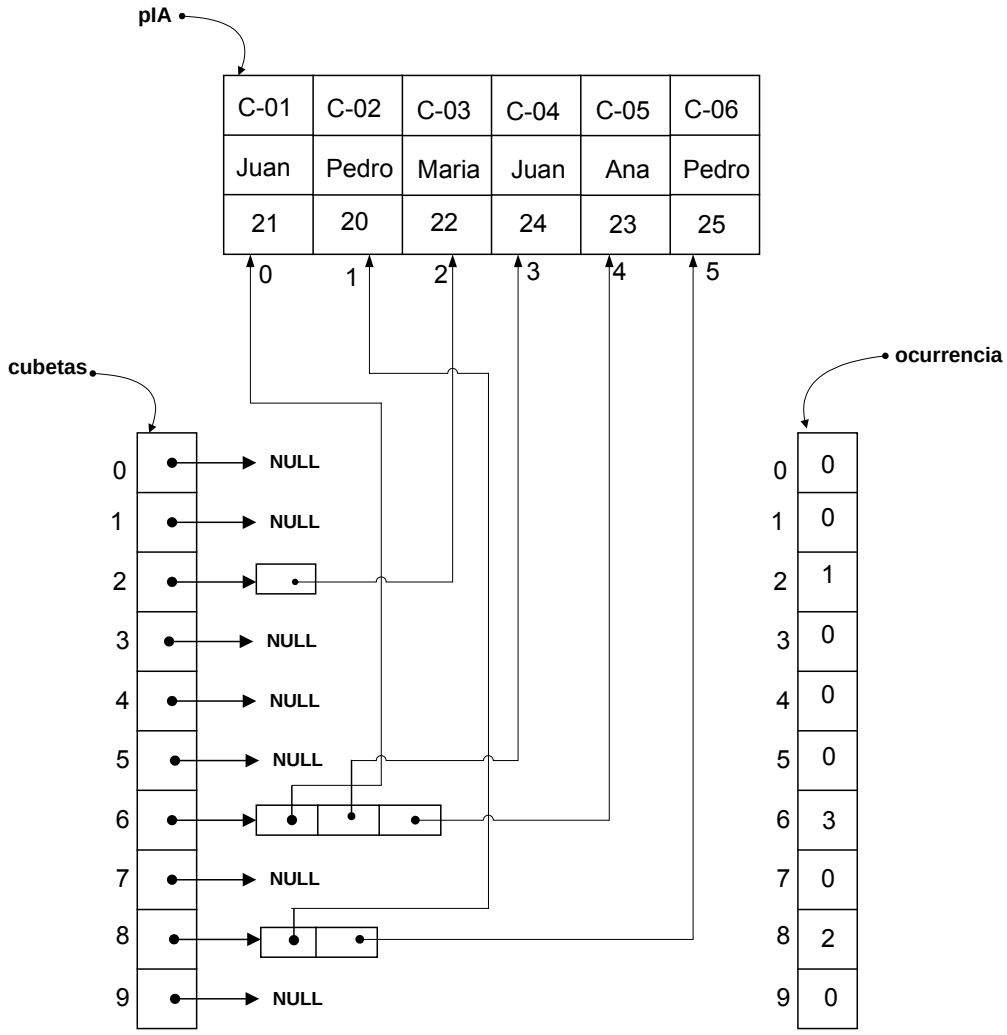


Ilustración 6-86: Estructuras de datos usadas en la técnica HASHING

pIA: Un puntero a un arreglo de estudiantes. Este arreglo contiene a todos los estudiantes introducidos por el usuario.

cubetas: Es un triple puntero a elementos de tipo info_alumno. El objetivo de este arreglo será apuntar a alumnos de acuerdo al peso generado. Por ejemplo, el alumno Juan con carnet c-01 (que ocupa la primera posición en el arreglo apuntado por pIA), tiene un peso de 6. Ya que Juan es el primer nombre con ese peso, se asignará memoria para un elemento referenciado por el índice 6 del arreglo cubetas. El siguiente elemento es el estudiante Pedro con carnet c-02, y resulta con un peso de 8; el mecanismo para apuntar será el mismo que el explicado para Juan. Este procedimiento se sigue para todos los estudiantes del arreglo referenciado por pIA. Cuando se encuentra a un estudiante que, el peso sea igual a uno ya apuntado, entonces, se reasignaría memoria para un elemento más para el índice correspondiente con ese peso en el arreglo cubetas. Así, el estudiante Juan con carnet c-04, también tiene un peso de 6; ya que previamente se había generado este peso, entonces se reasigna memoria para un elemento más desde el índice 6 del arreglo cubetas. Un tratamiento igual se sigue para el estudiante Ana.

ocurrencia: Mediante los contenidos de los índices de este arreglo, se llevan un conteo de la cantidad de ocurrencias de los nombres para cada índice. Esto se hace con el objetivo de, al momento de imprimir las cubetas, mostrar todos los nombres que tengan como peso, ese índice asociado.

ALGORITMOS DE LAS PRINCIPALES FUNCIONES

Función CalcularPeso

```
Function CalcularPeso (char *nom) int
Var let, i, suma integer
Para i ←0, hasta tamaño de cadena, hacer
        Convertir a mayúscula la cadena, si procede
        [Calcular el valor en el mapa de letras]
        let ←let – 64
        [Acumular la suma]
        suma ← suma + let
FinPara
Regresar suma MOD No_CUBETAS
FinFunction
```

Function main() void

```
var  i ← 0, no_alumnos ← 0, peso_nombre: integer;
var  j ←0, ocurrencias[No_CUBETAS]: integer;


var  pIA: pointer to Info_Alumno;
var cubetas: triple pointer to Info_Alumno;

pIA ←Asignar memoria para no_alumnos de tipo Info_Alumno
```

```
Leer datos para no_alumnos
Mostrar datos de no_alumnos

[Asignar memoria para el array cubetas]
cubetas ← Asignar memoria para No_CUBETAS de tipo Info_Alumno

[Calcular los pesos del nombre de los alumnos]
Para i ← 0, hasta no_alumnos, hacer
        peso_nombre ← CalcularPeso (pIA[i].nombre)
        [El valor del peso calculado está entre 0 y 9]
        Si (cubetas[peso_nombre] = NULL), entonces
                cubetas[peso_nombre] ← Asignar memoria para un elemento de tipo
                                       Info_Alumno
                cubetas[peso_nombre][0] ← &pIA[i]
                 Incrementar  ocurrencias[peso_nombre]
        FinSi
        SiNo    [Ya hay al menos un elemento apuntado]
                Incrementar  ocurrencias[peso_nombre]
                [Reasignar memoria para alojar al apuntador del nuevo elemento]
                cubetas[peso_nombre] ← Aumentar en un factor de
                                       ocurrencias[peso_nombre],  la memoria
                                       referenciada por cubetas[peso_nombre]
                Si (cubetas[peso_nombre] = NULL)
                        Error()
                j ← ocurrencias[peso_nombre] - 1
                cubetas[peso_nombre][j] ← &pIA[i]
        FinSiNo
FinPara

[Imprimir los índices (numeros de las cubetas) con sus respectivos nombres]
Para i ← 0, hasta No_CUBETAS, hacer
        Si (cubetas[i] = NULL)
                Imprimir “NO HAY NOMBRES ASOCIADOS A ESTA CUBETA ."
        SiNo    [ Al menos hay un nombre asociado]
                Para j ← 0, hasta ocurrencias[i], hacer
                        Imprimir (" %s ", cubetas[i][j]->nombre)
        FinSiNo
[Liberar la memoria asignada a las cubetas]
Para i ← 0, hasta No_CUBETAS, hacer
        Si (cubetas[i] <> NULL)
        Liberar cubetas[i]
FinPara

Liberar cubetas
Liberar pIA
FinFunction
```

ALGORITMOS HASH

Los algoritmos **hash** son métodos de búsqueda, que proporcionan una **"longitud de búsqueda"** pequeña y una flexibilidad superior a la de otros métodos, como puede ser, el método de "búsqueda binaria" que requiere que los elementos del array estén ordenados.

Por **"longitud de búsqueda"** se entiende el número de accesos que es necesario efectuar sobre un array para encontrar el elemento deseado.

Este método de búsqueda permite, como operaciones básicas, además de la búsqueda de un elemento, insertar un nuevo elemento (si el array está vacío, crearlo) y eliminar un elemento existente.

Arrays hash

Un array producto de la aplicación de un algoritmo **hash** se denomina **"array hash"** y son los arrays que se utilizan con mayor frecuencia en los sistemas de acceso. Gráficamente estos arrays tienen la forma mostrada en la Ilustración 6-87:

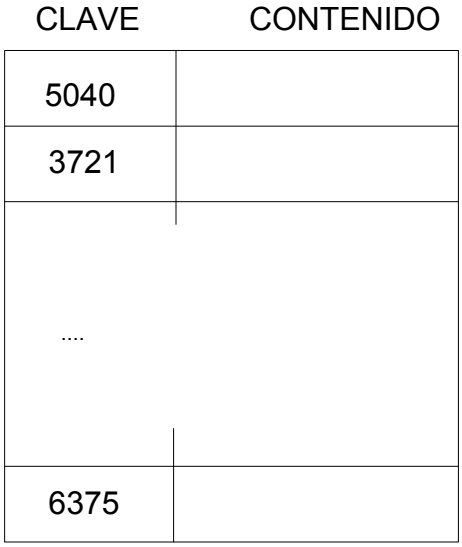


Ilustración 6-87: Representación gráfica de un array hash

El array se organiza con elementos formados por dos miembros: **clave** y **contenido.**

La **clave** constituye el medio de acceso al array. Aplicando a la **clave** una función de acceso **"fa"**, previamente definida, obtenemos un número entero positivo **"i"**, que nos da la posición del elemento correspondiente, dentro del array.

$$i = fa\ (clave)$$

Conociendo la posición tenemos acceso al **contenido.** El miembro **contenido** puede albergar directamente la información, o bien un puntero a dicha información, cuando ésta sea muy extensa.
El acceso, tal cual lo hemos definido, recibe el nombre de **"acceso directo".**

Como ejemplo, suponer que la **clave** de acceso se corresponde con el número del documento nacional de identidad **(dni)** y que el contenido son los datos correspondientes a la persona que tiene ese **dni** una función de acceso, **i = fa (dni)**, que haga corresponder la posición del elemento en el array con el **dni,** es inmediata:

$$i = dni$$

la cual da lugar a un acceso directo. Esta función así definida presenta un inconveniente y es que el número de valores posibles de **"i"** es demasiado grande para utilizar un array de este tipo.

Para solucionar este problema, es posible siempre, dado un array de longitud L, crear una **"función de acceso", fa,** de forma que nos genere un valor comprendido entre 0 y L, más comúnmente entre 1 y L.

En este caso puede suceder que dos o más claves den un mismo valor de **"i"**:

$$i = fa\ (clave_1) = fa\ (clave_2)$$

El método **hash** está basado en esta técnica; el acceso al array es directo por el número **"i"** y cuando se produce una **colisión** (dos claves diferentes dan un mismo número **"i"**) este elemento se busca en una zona denominada **"área de overflow".**

Método hash abierto

Este es uno de los métodos más utilizados. El algoritmo para acceder a un elemento del array de longitud L, es el siguiente:

1. Se calcula **i = fa (clave).**

2. Si la posición **"i"** del array está libre, se inserta la **clave** y el **contenido.** Si no está libre y la **clave** es la misma, error: clave duplicada. Si no está libre y la **clave** es diferente, incrementamos **"i"** en una unidad y repetimos el proceso descrito en este punto 2.

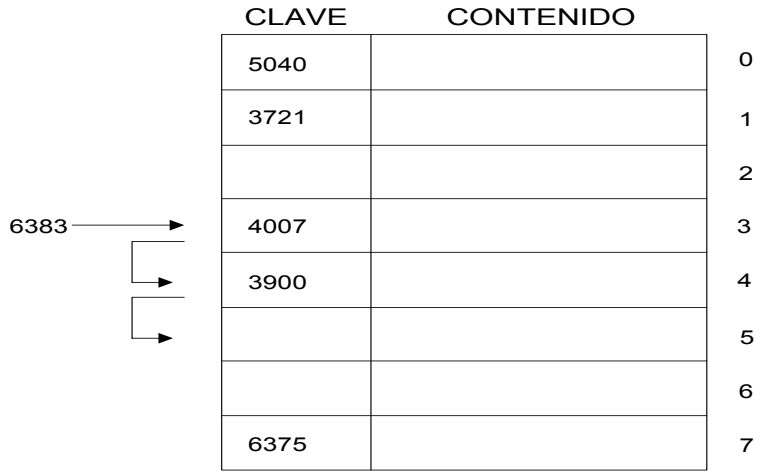


Ilustración 6-88: Inserción de un elemento en la técnica HASH abierto

En la Ilustración 6-88 observamos que, queremos insertar la clave 6383. Supongamos que aplicando la función de acceso obtenemos un valor de 3, esto es,

$$i = fa\ (6383) = 3$$

Como la posición 3 está ocupada y la clave es diferente, tenemos que incrementar **"i"** y volver de nuevo al punto 2 del algoritmo.

La **"longitud media de búsqueda"** en un **"array hash abierto"** viene dado por la expresión:

$$Accesos = (2-k) / (2-2k)$$

Siendo **k** igual al nº de elementos existentes en el array dividido por L.

Ejemplo:

Si existen 60 elementos en un array de longitud L = 100, el número medio de accesos para localizar un elemento será:

accesos = (2 – 60 / 100) / (2 – 2 * 60 / 100) = 1,75

En el método de "búsqueda binaria", el número de accesos viene dado por $\log_2 N$, siendo N el número de elementos del array.

Para reducir al máximo el número de colisiones y, como consecuencia, obtener una **"longitud media de búsqueda"** baja, es importante elegir bien la función de acceso.

Una **"función de acceso"** o **"función hash"** bastante utilizada y que proporciona una distribución de las claves uniforme y aleatoria es la **"función mitad del**

cuadrado" que dice: dada una clave C, se eleva al cuadrado (C^2) y se cogen **n** bits del medio, siendo $\mathbf{2^n <= L}$.

Ejemplo:

Supongamos:

L = 256 lo que implica n = 8
C = 625
C^2 = 390625 ($0 <= C^2 <= 2^{32} - 1$)
390625_{10} = $00000000000001011111010111100001_2$
n bits del medio: $01011111_2 = 95_{10}$

Otra **"función de acceso"** muy utilizada es la **"función módulo"** (resto de una división entera):

i = clave **módulo** L

Cuando se utilice esta función, es importante elegir un número primo para L, con la finalidad de que el número de colisiones sea pequeño.

Método hash con overflow

Una alternativa al método anterior es la de disponer de otro array separado, para insertar las claves que producen colisión, denominado**"array de overflow"**, en el cual se almacenan todas estas claves de forma consecutiva. La Ilustración 6-89, muestra esta técnica.

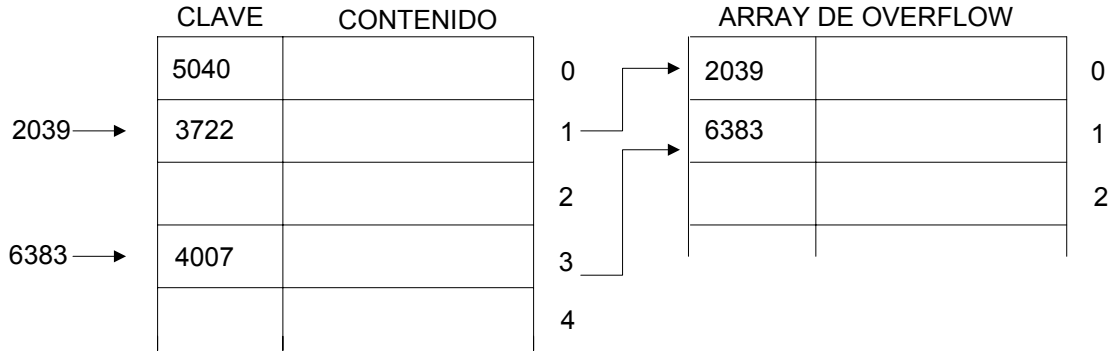


Ilustración 6-89: Estructura del método HASH con overflow

Otra forma alternativa más normal es organizar una lista encadenada por cada posición del array donde se produzca una colisión. La Ilustración 6-90, muestra esta variante.

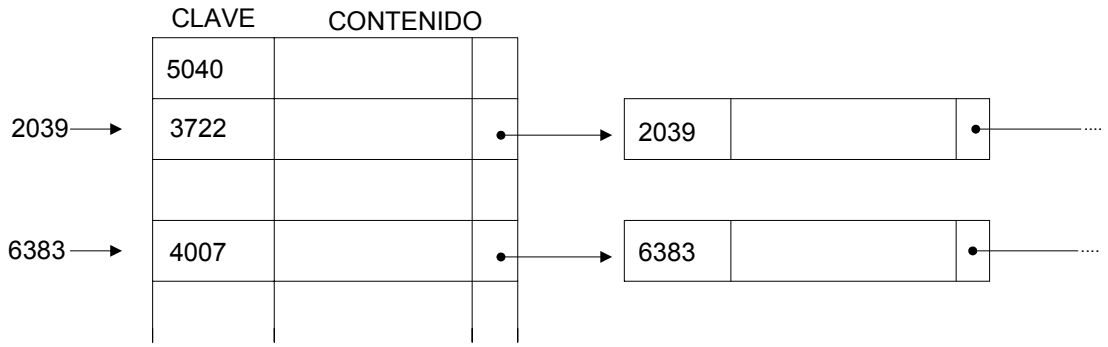


Ilustración 6-90: Otra alternativa del método HASH con overlow

Cada elemento de esta estructura incorpora un nuevo miembro P, el cual es un puntero a la lista encadenada de overflow.

Eliminación de elementos

En el método **hash** la eliminación de un elemento no es tan simple como dejar vacío dicho elemento, ya que esto da lugar a que los elementos insertados por colisión no puedan ser accedidos. Por ello se suele utilizar un miembro (campo) complementario que sirva para poner una marca de que dicho elemento está eliminado. Esto permite acceder a otros elementos que dependen de él por colisiones, ya que la clave se conserva y también permite insertar un nuevo elemento en esa posición cuando se dé una nueva colisión.

UN EJEMPLO DE UN ARRAY HASH CON UNA FUNCIÓN DE ACCESO MÓDULO

Enunciado: Crear un array **hash** de una determinada longitud **L** que permita almacenar los datos número de **matricula** y **nombre** de los alumnos matriculados en una cierta Universidad, utilizando el método **hash abierto** y la función de acceso **módulo.**

El pseudocódigo para el método **hash** abierto es el siguiente:

```
< función hash (array, n_elementos, elemento x) >
        [El array está inicializado a valor 0]
        i = matricula módulo n_elementos
        DO WHILE (no insertado y haya elementos libres)
                IF (elemento “i” está libre) THEN
                        Copiar elemento x  en la posición i
                ELSE
                        IF (clave duplicada) THEN
                                error: clave duplicada
                        ELSE
                                [se ha producido una colisión]
                                [avanzar al siguiente elemento]
                                i = i + 1
```

```
                                IF (i = n_elemento) THEN
                                        i = 0
                                ENDIF
                        ENDIF
                ENDIF
        ENDDO
END [hash]
```

A continuación se muestra el programa correspondiente a éste método. El programa se ha dividido en 2 partes (archivo de cabecera y programa principal)

/* globales.h */

```
/* Implementación de la técnica HASH SIMPLE */
typedef struct datos elemento;                  //nuevo tipo de elemento

struct datos                                            // elemntos para el array
{
        unsigned int matricula;         // matricula del estudiante
        char *nombre;                   // nombre del estudiante

};

int tam_registro = sizeof (elemento);
#define N 81   /* Tamaño máximo de un nombre */

// Declaración de funciones prototipos
void error (void);
void hash (elemento *, int, elemento);
int  siguiente_primo (int);
void ImprimirHASH (elemento *, int);
void InicializarHASH (elemento *, int);
void ManipularHASH (elemento *, int);
void GrabarArrayHASH (elemento *, int, FILE *);


////////////////// rutina de maipulación del error ////////////////////
void error (void)
{
        perror ("error: insuficiente espacio de memoria");
        exit(1);
}

////////////// método hash abierto ///////////////////////
void hash (elemento *a, int n_elementos, elemento x)
{
        int i;     // indice para referenciar un elemento
        int conta = 0;  // contador
        bool insertado = false;         // clave ya insertada?
```

```
        // 1er paso del algoritmo
        i = x.matricula % n_elementos;        // Función de acceso módulo

        // Paso 2 del algoritmo
        while (insertado == false && conta < n_elementos)
        {
                if (a[i].matricula == 0) // elemento libre
                {
                        // Copiar el elemento x a a[i], elemento a elemento
                        a[i].matricula = x.matricula;

                        // Asignar memoria para el nombre
                        a[i].nombre = (char *) malloc ( (strlen(x.nombre) + 1) * sizeof(char));
                        if (a[i].nombre == NULL)
                                error();

                        strcpy(a[i].nombre, x.nombre);

                        insertado = true;
                }

                // si no está libre, entonces hay una colisión
                else    // 1 caso: clave duplicada
                {
                        if (x.matricula == a[i].matricula)
                        {
                                printf("\n\t Error: matricula duplicada \n");
                                insertado = true;
                        }

                        else    // 2 caso: colisión
                        {
                                // siguiente elemento libre
                                i++;
                                conta++;

                                if (i == n_elementos)
                                        i = 0;

                        }
                } // fin del else

        } // fin del while

        if (conta == n_elementos)
                printf("\n\t Error: array lleno \n");

}

////////// Buscar un número primo a partir de un número dado//////////////
int siguiente_primo (int n)
```

```
{
	bool primo = false;
	int i;

	if (n % 2 == 0)// El número pasado es par?
		n++;

	while (primo == false)
	{
		primo = true;   // primo
		for (i = 3; i <= (int) sqrt ((double)n); i += 2)
		{
			if (n % i == 0)
				primo = false;  // no primo

			if (primo == false)
				n += 2;// siguiente impar
		} // fin del for
	}

	return n;
}

// Mostrar los elementos del array HASH
void ImprimirHASH (elemento *a, int ne)
{
	int i;

	printf ("\n\n\t ELEMENTOS DEL ARRAY HASH...");
	printf ("\n\t--------------------------------\n");

	for (i = 0; i < ne; i++)
	{
		printf ("\n\t ELEMENTO %d del array", i);
		printf("\n\t Matricula: %u", a[i].matricula);
		printf("\n\t Nombre: %s", a[i].nombre);
		printf ("\n");

	}

}

// Función InicializaHASH
void InicializarHASH (elemento *a, int n_elementos)
{
	int i;

	// inicializar el array
	for (i = 0; i < n_elementos; i++)
	{
		a[i].matricula = 0;
```

```
                a[i].nombre = NULL;

        }
}


// Introducir datos en el array HASH
void ManipularHASH (elemento *a, int n_elementos)
{
        char nom[81];
        int i;
        elemento x;


        i = 0;
        // introducir datos
        printf("\n\t Introducir datos. Finalizar con matricula = 0 \n\n");
        printf ("\n\t ALUMNO No %d", i + 1);

        printf("\n\t Matricula: ");
        scanf("%u", &x.matricula);

        fflush(stdin);

        while (x.matricula != 0)
        {
                printf ("\t Nombre: ");
                gets (nom);

                // Asignar espacio para nombre
                x.nombre = (char *) malloc (strlen (nom) + 1);

                if (!x.nombre)
                        error ();

                strcpy (x.nombre, nom);

                hash (a, n_elementos, x);      // Llamada a la función hash

                i++;
                printf ("\n\t ALUMNO No %d", i + 1);
                printf("\n\t Matricula: ");
                scanf("%u", &x.matricula);
                fflush(stdin);

                free (x.nombre);
        } // fin del while

} // Fin de ManipularHASH()

// Escribir el array HASH de n_elementos al disco
```

```
void GrabarArrayHASH (elemento *a, int n_elementos, FILE *pf)
{
        int i, j;
        int tam_nombre;

        pf = fopen ("Array_HASH", "wb");

        if (pf == NULL)
        {
                perror ("No se pudo abrir el fichero");
                exit (0);
        }

        rewind (pf);

        for (i = 0; i < n_elementos; i++)
        {
                /* Si el registro que se va a escribir en el disco duro, no está vacío,
                   entonces, a continuación escribimos el nombre con la función
                   fputs. Esto se hace así, ya que al ser el nombre un miembro que
                   es un puntero, la función fwrite escribe solamente la dirección
                   de comienzo de la cadena de caracteres que representa al nombre, y
                   no el nombre en si. Para los registros vacíos, NO se escribirá la
                   cadena de caracteres a continuación del registro.
                */

                /* Verificamos si el registro a escribir no está vacío */

                if (a[i].matricula != 0)
                {
                        /* Escribir en el disco un registro con datos validos */
                        fwrite (&a[i], tam_registro, 1, pf);

                        tam_nombre = strlen (a[i].nombre) + 1;
                        putw (tam_nombre, pf);

                        for (j = 0; j < tam_nombre; j++)
                                fputc (a[i].nombre[j], pf);

                } // cierre del if (a[i].matricula != 0)

        } // cierre del for(i = 0; i < n_elementos; i++)

        fclose (pf);

}
```

/* principal.cpp */

```
/****************** Arrays Hash ***************/
/* Crear un array hash de una determinada longitud L que permita almacenar los datos:
   número de matricula y nombre de los alumnos matriculados en una cierta Universidad,
   utilizando el método hash abierto y la función de acceso módulo.*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <crtdbg.h>
#include "globales.h"

void main ()
{
        elemento *a;            // dirección de comienzo del array
        int n_elementos;        // Número de elementos del array
        int i, j;
        FILE *pf, *pf1;         // Punteros a ficheros
        elemento registro;      // tipo elemento
        // char buffer[N];                  // almacena el nombre
        int long_nombre;        // longitud del nombre

        // Averiguar si el array hash ya existe en el disco duro....
        pf =  fopen ("Array_HASH", "rb");
        pf1 = fopen ("cantidad", "rb");

        if (pf != NULL)         // ¿Existe el fichero?
        {
                // El fichero ya existe, reconstruir la lista en memoria principal
                printf ("\n\t !!!ATENCION!!! El fichero ya existe.");
                printf ("\n\t Trayendo datos del disco....");

                // Traer la cantidad de elementos desde el disco
                n_elementos = getw (pf1);

                // Crear el array dinámico "a"
                a = (elemento *) malloc (n_elementos * sizeof (elemento));
                if(!a)
                        error ();

                InicializarHASH (a, n_elementos);

                // Reconstruir el array HASH
                i = 0;
                rewind (pf);
                fread (&registro, tam_registro, 1, pf);

                while (!feof (pf) && !ferror (pf))
                {
                        long_nombre = getw (pf);
```

```
                registro.nombre = (char *) malloc (long_nombre * sizeof (char));

                if (registro.nombre == NULL)
                        error ();

                // Copiamos caracter a caracter el nombre
                for (j = 0; j < long_nombre; j++)
                        registro.nombre[j] = fgetc (pf);

                // Una vez que ya tengo el registro creado en MP, llamar a la
                // funcion HASH para ubicarlo en el array
                hash (a, n_elementos, registro);       // Llamada a la función hash

                // Liberamos la memoria para el nombre del registro
                free (registro.nombre);

                fread (&registro, tam_registro, 1, pf);
                i++;

        } // cierre del while (!feof (pf) && !ferror (pf))

        printf ("\n\t Datos traidos del disco....\n\n");

        // Imprimimos el array traido del disco
        printf ("\n\t El array traido del disco es....\n\n");
        ImprimirHASH (a, n_elementos);

        // Trabajar con el array HASH creado
        ManipularHASH (a, n_elementos);

        fclose (pf);      // Cerrar el fichero para luego grabar en el disco
        fclose (pf1);

} // cierre del if (pf != NULL)


//-------------------------------------------------------
else
{
        // El fichero no existe, crear el array
        printf ("\n\t EL FICHERO NO EXISTIA Y SE HA CREADO...");

        printf ("\n\t CONSTRUCCION DE UN ARRAY HASH...");
        printf("\n\t Numero de elementos: ");
        scanf("%d", &n_elementos);

        n_elementos = siguiente_primo (n_elementos);

        printf ("\n\t El sig primo despues de n es: %d", n_elementos);
```

```
                // Crear el array dinámico "a"
                a = (elemento *) malloc (n_elementos * sizeof (elemento));

                if (!a)
                        error ();

                InicializarHASH (a, n_elementos);

                // Trabajar con el array HASH
                ManipularHASH (a, n_elementos);

                // 1. Escribir la cantidad de elementos del array al disco
                pf1 = fopen ("cantidad", "wb");
                putw (n_elementos, pf1);
                fclose (pf1);


        } // cierre del else

        // Imprimir el arreglo hash creado
        printf ("\n\t El contenido del array es....");
        ImprimirHASH (a, n_elementos);


        printf ("\n\t Escribiendo el contenido del array al disco...\n");
        // Escribir el array HASH al disco
        GrabarArrayHASH (a, n_elementos, pf);


        // Liberar la memoria asignada para el array a
        // Liberar primero la memoria asignada a los nombres
        for (i = 0; i < n_elementos; i++)
        {
                if (a[i].nombre != NULL)
                        free (a[i].nombre);
        }

        free (a);

        if (_CrtDumpMemoryLeaks())
                printf ("\n\t Se encontraron LAGUNAS DE MEMORIA...");
}
```

SALIDA DEL PROGRAMA

A continuación se muestra la salida del programa, ejecutando el programa dos veces. Igual que antes, las entradas de datos por parte del usuario, están mostradas en negrita y subrayadas.

PRIMERA EJECUCIÓN

La primera ejecución (probando con valores de **clave menores que la longitud del array**), se muestra en la Ilustración 6-91.

```
EL FICHERO NO EXISTIA Y SE HA CREADO...
CONSTRUCCION DE UN ARRAY HASH...
Numero de elementos: 20

El sig primo despues de n es: 23
Introducir datos. Finalizar con matricula = 0

ALUMNO No 1
Matricula: 10
Nombre: Juan

ALUMNO No 2
Matricula: 12
Nombre: Pedro

ALUMNO No 3
Matricula: 8
Nombre: Maria

ALUMNO No 4
Matricula: 17
Nombre: Ana

ALUMNO No 5
Matricula: 0

El contenido del array es....

ELEMENTOS DEL ARRAY HASH...
--------------------------------

ELEMENTO 0 del array
Matricula: 0
Nombre: (null)

ELEMENTO 1 del array
Matricula: 0
Nombre: (null)

ELEMENTO 2 del array
Matricula: 0
Nombre: (null)
```

```
ELEMENTO 3 del array
Matricula: 0
Nombre: (null)

ELEMENTO 4 del array
Matricula: 0
Nombre: (null)

ELEMENTO 5 del array
Matricula: 0
Nombre: (null)

ELEMENTO 6 del array
Matricula: 0
Nombre: (null)

ELEMENTO 7 del array
Matricula: 0
Nombre: (null)

ELEMENTO 8 del array
Matricula: 8
Nombre: Maria

ELEMENTO 9 del array
Matricula: 0
Nombre: (null)

ELEMENTO 10 del array
Matricula: 10
Nombre: Juan

ELEMENTO 11 del array
Matricula: 0
Nombre: (null)

ELEMENTO 12 del array
Matricula: 12
Nombre: Pedro

ELEMENTO 13 del array
Matricula: 0
Nombre: (null)

ELEMENTO 14 del array
Matricula: 0
Nombre: (null)

ELEMENTO 15 del array
Matricula: 0
Nombre: (null)

ELEMENTO 16 del array
Matricula: 0
Nombre: (null)
```

```
        ELEMENTO 17 del array
        Matricula: 17
        Nombre: Ana

        ELEMENTO 18 del array
        Matricula: 0
        Nombre: (null)

        ELEMENTO 19 del array
        Matricula: 0
        Nombre: (null)

        ELEMENTO 20 del array
        Matricula: 0
        Nombre: (null)

        ELEMENTO 21 del array
        Matricula: 0
        Nombre: (null)

        ELEMENTO 22 del array
        Matricula: 0
        Nombre: (null)

        Escribiendo el contenido del array al disco...
Press any key to continue
```

Ilustración 6-91: Primera ejecución de la técnica HASHING con función de acceso módulo

SEGUNDA EJECUCIÓN

La segunda ejecución (probando con valores de **clave mayores que la longitud del array**), se muestra en la Ilustración 6-91.

```
!!!ATENCION!!! El fichero ya existe.
Trayendo datos del disco....
Datos traidos del disco....

El array traido del disco es....

ELEMENTOS DEL ARRAY HASH...
---------------------------------
ELEMENTO 0 del array
Matricula: 0
Nombre: (null)

ELEMENTO 1 del array
Matricula: 0
Nombre: (null)

ELEMENTO 2 del array
Matricula: 0
Nombre: (null)

ELEMENTO 3 del array
Matricula: 0
Nombre: (null)

ELEMENTO 4 del array
Matricula: 0
Nombre: (null)

ELEMENTO 5 del array
Matricula: 0
Nombre: (null)

ELEMENTO 6 del array
Matricula: 0
Nombre: (null)

ELEMENTO 7 del array
Matricula: 0
Nombre: (null)

ELEMENTO 8 del array
Matricula: 8
Nombre: Maria

ELEMENTO 9 del array
Matricula: 0
Nombre: (null)

ELEMENTO 10 del array
Matricula: 10
Nombre: Juan
```

ELEMENTO 11 del array
Matricula: 0
Nombre: (null)

ELEMENTO 12 del array
Matricula: 12
Nombre: Pedro

ELEMENTO 13 del array
Matricula: 0
Nombre: (null)

ELEMENTO 14 del array
Matricula: 0
Nombre: (null)

ELEMENTO 15 del array
Matricula: 0
Nombre: (null)

ELEMENTO 16 del array
Matricula: 0
Nombre: (null)

ELEMENTO 17 del array
Matricula: 17
Nombre: Ana

ELEMENTO 18 del array
Matricula: 0
Nombre: (null)

ELEMENTO 19 del array
Matricula: 0
Nombre: (null)

ELEMENTO 20 del array
Matricula: 0
Nombre: (null)

ELEMENTO 21 del array
Matricula: 0
Nombre: (null)

ELEMENTO 22 del array
Matricula: 0
Nombre: (null)

Introducir datos. Finalizar con matricula = 0

ALUMNO No 1
Matricula: ***50***
Nombre: ***Lisseth***

ALUMNO No 2
Matricula: ***120***
Nombre: ***Marisol***

ALUMNO No 3
Matricula: ***80***
Nombre: ***Paola***

ALUMNO No 4
Matricula: ***110***
Nombre: ***Ricardo***

ALUMNO No 5
Matricula: 0

El contenido del array es....

ELEMENTOS DEL ARRAY HASH...

ELEMENTO 0 del array
Matricula: 0
Nombre: (null)

ELEMENTO 1 del array
Matricula: 0
Nombre: (null)

ELEMENTO 2 del array
Matricula: 0
Nombre: (null)

ELEMENTO 3 del array
Matricula: 0
Nombre: (null)

ELEMENTO 4 del array
Matricula: 50
Nombre: Lisseth

ELEMENTO 5 del array
Matricula: 120
Nombre: Marisol

ELEMENTO 6 del array
Matricula: 0
Nombre: (null)

ELEMENTO 7 del array
Matricula: 0
Nombre: (null)

ELEMENTO 8 del array
Matricula: 8
Nombre: Maria

```
ELEMENTO 9 del array
Matricula: 0
Nombre: (null)

ELEMENTO 10 del array
Matricula: 10
Nombre: Juan

ELEMENTO 11 del array
Matricula: 80
Nombre: Paola

ELEMENTO 12 del array
Matricula: 12
Nombre: Pedro

ELEMENTO 13 del array
Matricula: 0
Nombre: (null)

ELEMENTO 14 del array
Matricula: 0
Nombre: (null)

ELEMENTO 15 del array
Matricula: 0
Nombre: (null)

ELEMENTO 16 del array
Matricula: 0
Nombre: (null)

ELEMENTO 17 del array
Matricula: 17
Nombre: Ana

ELEMENTO 18 del array
Matricula: 110
Nombre: Ricardo

ELEMENTO 19 del array
Matricula: 0
Nombre: (null)

ELEMENTO 20 del array
Matricula: 0
Nombre: (null)

ELEMENTO 21 del array
Matricula: 0
Nombre: (null)

ELEMENTO 22 del array
Matricula: 0
Nombre: (null)
```

```
      Escribiendo el contenido del array al disco...
Press any key to continue
```

Ilustración 6-92: Segunda ejecución de la técnica HASHING con función de acceso módulo

EXPLICACIÓN DE LA CODIFICACIÓN DEL PROGRAMA

La ubicación de los elementos dentro del arreglo, cuando se introducen los alumnos con claves: 10, 12, 8 y 17 (en este orden), se muestra en la Ilustración 6-93. A como puede observarse, todos estos valores de clave, son menores que la longitud del array, por lo cual, al aplicar la función de acceso módulo (**i = clave % Longitud del arreglo**), la posición dentro del arreglo da la misma clave.

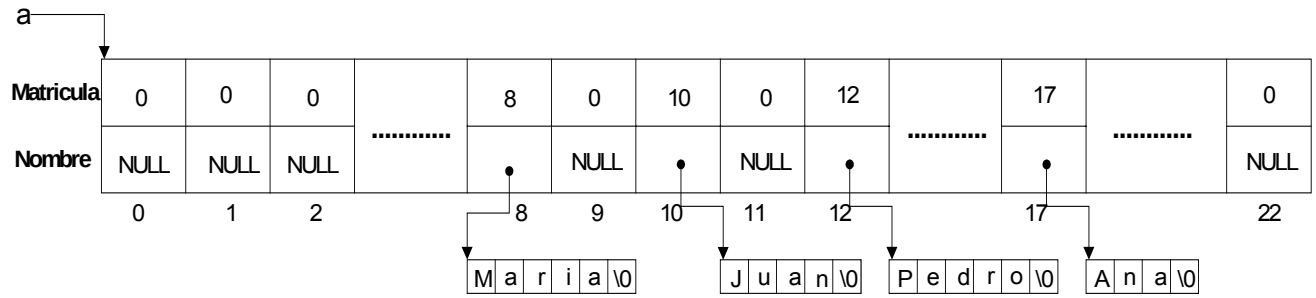


Ilustración 6-93: Ubicación de elementos: Primera ejecución de la técnica HASHING con función de acceso módulo

En el caso de la segunda ejecución, los valores de clave son mayores que la longitud del arreglo, de esta manera, la ubicación dentro del arreglo, dependerá del valor de clave misma. Por ejemplo para el caso del alumno Ricardo, con clave igual 110, su ubicación dentro del arreglo sería: i = 110 % 17 → i = 8. La Ilustración 6-94, muestra lo expuesto.

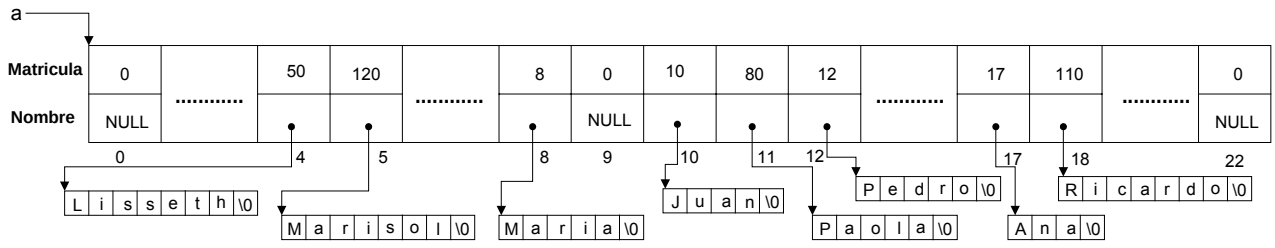


Ilustración 6-94: Ubicación de elementos: Segunda ejecución de la técnica HASHING con función de acceso módulo

ALGORITMOS DE LAS PRINCIPALES FUNCIONES

```
register elemento
{
        matricula : integer;            // matricula del estudiante
        nombre: pointer to char;        // nombre del estudiante

};

var tam_registro ← FuncionObtenerTamanyo(register elemento)

Function main( ) void
var a: pointer to elemento
var i, n_elementos: integer
var registro: register elemento
var nombre: char[81]
var pf, pf1: pointer to FILE
var long_nombre: integer

[Abrir el archivo Array_HASH en modo binario para lectura]
pf  ← AbrirFichero (Array_HASH)
pf  ← AbrirFichero (cantidad)

Si el fichero Array_HASH existe, entonces
        Imprimir "EL FICHERO EXISTE…. TRAYENDO DATOS DEL DISCO"

        [Leer dato del fichero cantidad]
        n_elementos ← LecturaDeDatoEntero (pf1)

        [Crear el array dinámico]
        a ← Asignar memoria para n_elementos de tipo elemento

        [Reconstruir el array HASH]
        LeerRegistro (direccion_var_registro, tam_registro, 1, pf)

        Mientras (no se llegue al fin de archivo o no haya error en la lectura del fichero
                apuntado por pf), hacer
                [Leer el dato entero almacenado a continuación del registro leído]
                long_nombre ← LecturaDeDatoEntero (pf)

                [Asignar memoria para registro.nombre de tamaño long_nombre]
                registro.nombre  ←  AsignarMemoria (long_nombre)

                Si no se pudo asignar memoria, entonces
                        Error()

                [Copiamos caracter a caracter el nombre]
                Para j ←  0 hasta  long_nombre, hacer
                        registro.nombre[j] = fgetc (pf);

                [Una vez que ya tengo el registro creado en MP, llamar a la funcion HASH
                para ubicarlo en el array]
                hash (a, n_elementos, registro)
```

```
                [Liberamos la memoria para el nombre del registro]
                LiberarMemoria (registro.nombre)

                LeerRegistro (direccion_var_registro, tam_registro, 1, pf)
                i ← i + 1

        FinMientras

        Imprimir " Datos traidos del disco....
        [Imprimimos el array traido del disco]
        Imprimir “El array traido del disco es....”
        ImprimirHASH (a, n_elementos)

        [Trabajar con el array HASH creado]
        ManipularHASH (a, n_elementos)

        [Cerrar los ficheros]
        CerrarFichero (pf)
        CerrarFichero (pf1)
FinSi  [Si el fichero Array_HASH existe, entonces]


SiNo
        [El fichero no existe, crear el array]
        Imprimir “EL FICHERO NO EXISTIA Y SE HA CREADO"

        Imprimir “CONSTRUCCION DE UN ARRAY HASH.."
        [Pedir al usuario cantidad de elementos que formarán el array]
        Imprimir  “Numero de elementos: “
        LecturaDeDatoEntero (n_elementos)

        [Obtener el siguiente primo]
        n_elementos ←  siguiente_primo (n_elementos)

        [Crear el array dinámico "a"]
        a ←  AsignarMemoria (n_elementos de tipo register elemento)

        Si no se pudo asignar memoria, entonces
                Error()
        [Inicializar el array a, a valores vacíos]
        InicializarHASH (a, n_elementos)

        [Trabajar con el array HASH]
        ManipularHASH (a, n_elementos)

        [Escribir la cantidad de elementos del array al disco]
        [Abrir el fichero cantidad para leer. Si no existe, crearlo]
        pf1 ← AbrirFichero (cantidad)
        FuncionEscribirDatoEntero (n_elementos, pf1)
        CerrarFichero (pf1)
FinSino
        [Imprimir el arreglo hash creado]
```

```
        Imprimir "El contenido del array es…"
        ImprimirHASH (a, n_elementos)

        Imprimir "Escribiendo el contenido del array al disco.."
        [Escribir el array HASH al disco]
        GrabarArrayHASH (a, n_elementos, pf)

        [Liberar la memoria asignada para el array a]
        [Liberar primero la memoria asignada a los nombres]
        Para i ← 0 hasta n_elementos, hacer
                Si (a[i].nombre <>  NULL), entonces
                LiberarMemoria (a[i].nombre)
        FinPara

        [Liberar la memoria asignada al array a]
        LiberarMemoria (a)

FinFunction [Fin de la function main]
```

Function ManipularHASH (a: pointer to register elemento, n_elementos: integer) void

```
var nom: char(81)
var i: integer
var x: register elemento


// Pedir matricula del primer estudiante
x.matricula ← Lectura de matrícula

Mientras x.matricula <> 0, hacer

        // Leer nombre y almacenarlo en nom
        nom ← Lectura del nombre

        // Asignar memoria para el miembro nombre de x, del tamaño de nom
        x.nombre  =  ← AsignarMemoria (longitud de nom)  +  1
        Si no ha sido posible asignar memoria para x.nombre, entonces
                Generar mensaje de Error
                Salir de la aplicación
        // Copiar el contenido de nom en x.nombre
        x.nombre ← nom
        // Llamado a la función hash
        hash (a, n_elementos, x)
        i ← i  +  1

        // Pedir matrícula
        x.matricula ← Lectura de matrícula

        // Liberar memoria para x.nombre
        LiberarMemoria (x.nombre)

FinMientras
FinFunction
```

2.Función hash ()

Fuction hash (elemento *a, int n_elementos, elemento x) void

```
var i, conta ← 0: integer
var insertado: booleano

[Función de acceso modulo]
i ← x.matricula MOD n_elementos

Mientras insertado = false y conta < n_elementos, entonces
        Si a[i].matricula = 0, entonces
                [Elemento libre]
                [Copiar el elemento x a a[i], elemento a elemento]
                a[i].matricula  ←  x.matricula
                a[i].nombre ← AsignarMemoria(longitud de x.nombre + 1)
                Si no ha sido posible asignar memoria para a[i].nombre, entonces
                        Generar Mensaje de Error
                        Salir de la aplicación
                // Copiar el nombre que reside en x, en el campo nombre de a
                a[i].nombre ←  x.nombre
                insertado ← true
        FinSi

        SiNo está libre, entonces
                Si x.matricula = a[i].matricula, entonces [Clave duplicada]
                        Imprimir “Clave duplicada”
                        insertado ← true
                FinSi

                SiNo    [Colisión]
                        i ← i + 1          [Incrementar índice]
                        conta ← conta + 1       [El número de elementos de los que consta
                                                  el array, ha aumentado en uno]

                        Si i = n_elementos, entoces    [Se ha llegado al final del
                                                                arreglo?]
                                i ← 0
                FinSiNo
        FinSino
FinMientras

Si conta = n_elementos, entonces
        Imprimir “Array lleno”

FinFunction
```

UN EJEMPLO DE UN ARRAY HASH CON UNA FUNCIÓN DE ACCESO MITAD DEL CUADRADO

Enunciado: Crear un array **hash** de una determinada longitud **L** que permita almacenar los datos número de **matrícula** y **nombre** de los alumnos matriculados en una cierta Universidad, utilizando el método **hash abierto** y la función de acceso **mitad del cuadrado.**

A continuación se muestra el programa correspondiente a éste método. El programa se ha dividido en 2 partes (archivo de cabecera y programa principal)

/* globales.h */

```
/* Implementación de la técnica HASH SIMPLE */
typedef struct datos elemento;        //nuevo tipo de elemento

struct datos                          // elemntos para el array
{
        unsigned long matricula;      // matricula del estudiante
        char *nombre;                 // nombre del estudiante

};

// Tamaño de un registro
int tam_registro = sizeof (elemento);

// Número máximo de bits para representar la clave (matricula) al cuadrado
#define NUM_MAXIMO_BITS 32

// Declaración de funciones prototipos
void error (void);
void hash (elemento *, int, elemento);
void ImprimirHASH (elemento *, int);
int MitadDelCuadrado (int n_elementos, int matricula);
void ConvertirA_BinPuro (unsigned long matri_cuad, unsigned int *num_bin);
unsigned long ConvertirBP_A_Decimal(unsigned int *num_bin_mitad, int n);
void InicializarHASH (elemento *, int);
void ManipularHASH (elemento *, int);
void GrabarArrayHASH (elemento *, int, FILE *);


////////////////// rutina de maipulación del error ////////////////////
void error (void)
{
        perror ("error: insuficiente espacio de memoria");
        exit(1);
}

////////////// método hash abierto ///////////////////////
void hash (elemento *a, int n_elementos, elemento x)
```

```
{
        int i;     // indice para referenciar un elemento
        int conta = 0;   // contador
        bool insertado = false;          // clave ya insertada?

        // 1er paso del algoritmo
        // Función de acceso mitad del cuadrado
        // Pasarle a dicha función: la longitud del arreglo, y la clave
        i = MitadDelCuadrado (n_elementos, x.matricula);

        // Paso 2 del algoritmo
        while (insertado == false && conta < n_elementos)
        {
                if (a[i].matricula == 0) // elemento libre
                {
                        // Copiar el elemento x a a[i], elemento a elemento
                        a[i].matricula = x.matricula;

                        // Asignar memoria para el nombre
                        a[i].nombre = (char *) malloc ( (strlen(x.nombre) + 1) * sizeof(char));
                        if (a[i].nombre == NULL)
                                error();

                        strcpy(a[i].nombre, x.nombre);

                        insertado = true;
                }

                // si no está libre, entonces hay una colisión
                else    // 1 caso: clave duplicada
                {
                        if (x.matricula == a[i].matricula)
                        {
                                printf("\n\t Error: matricula duplicada \n");
                                insertado = true;
                        }

                        else    // 2 caso: colisión
                        {
                                // siguiente elemento libre
                                i++;
                                conta++;

                                if (i == n_elementos)
                                        i = 0;

                        }
                } // fin del else

        } // fin del while
```

```
        if (conta == n_elementos)
                printf("\n\t Error: array lleno \n");
}


// Mostrar los elementos del array HASH
void ImprimirHASH (elemento *a, int ne)
{
        int i;

        printf ("\n\n\t ELEMENTOS DEL ARRAY HASH...");
        printf ("\n\t--------------------------------\n");

        for (i = 0; i < ne; i++)
        {
                printf ("\n\n\t ELEMENTO %d DEL ARRAY....", i);
                printf ("\n\t Matricula: %u", a[i].matricula);
                printf ("\n\t Nombre: %s", a[i].nombre);
                printf ("\n");

        }

}

// Función de acceso mitad del cuadrado
int MitadDelCuadrado (int n_elementos, int matricula)
{
        int n, mitad, aux = 0, i;
        unsigned long matri_cuad, decimal;
        unsigned int *num_bin_mitad;
        unsigned int num_bin[NUM_MAXIMO_BITS];

        // 1. Calcular n bits del medio
        n = (int) ( log ( (double) n_elementos) / log( (double) 2));

        // 2. Elevar clave al cuadrado
        matri_cuad =  (unsigned long) pow ( (double) matricula, (double) 2);

        // Inicializar a cero el arreglo num_bin
        for (i = 0; i < NUM_MAXIMO_BITS; i++)
                num_bin[i] = 0;

        // 3. Convertir matri_cuad a binario puro
        ConvertirA_BinPuro (matri_cuad, num_bin);

        // 4. Obtener el indice mitad del arreglo num_bin
        mitad = (NUM_MAXIMO_BITS / 2);

        // 5. Asignar memoria para almacenar el binario de la mitad del numero
        num_bin_mitad = (unsigned int *) malloc (n * sizeof(unsigned int));
        if(num_bin_mitad == NULL)
```

```
                error();

        aux = mitad - (n / 2);

        if (aux < 0)
                decimal = 1;

        else
        {
                // Copiar los bits del medio a num_bin_mitad
                for (i = 0; i < n; i++)
                {
                        num_bin_mitad[i] = num_bin[aux];
                        aux++;
                }

                decimal = ConvertirBP_A_Decimal(num_bin_mitad, n);
        }

        free (num_bin_mitad);

        return (int)(decimal);
}

// Convertir un valor doble a binario puro
void ConvertirA_BinPuro (unsigned long matri_cuad, unsigned int *num_bin)
{
        int i = 0;

        if(matri_cuad == 1)
        {
                num_bin[0] = 1;
                return;          // Regresamos al main
        }


        i = 0;
        while (matri_cuad > 1)
        {
                num_bin[i] = (unsigned int) matri_cuad % 2;
                matri_cuad = matri_cuad / 2;
                i++;
        }

        num_bin[i] = (unsigned int) matri_cuad;
}

// Convertir de binario a decimal
unsigned long ConvertirBP_A_Decimal(unsigned int *num_bin_mitad, int n)
{
        unsigned long dec = 0;
```

```
        int i;

        for (i = 0; i < n; i++)
                dec = dec + (unsigned long )((double) num_bin_mitad[i] * pow ((double) 2
                                                        ,(double) i));

        return (dec);
}

// Función InicializaHASH
void InicializarHASH (elemento *a, int n_elementos)
{
        int i;

        // inicializar el array
        for (i = 0; i < n_elementos; i++)
        {
                a[i].matricula = 0;
                a[i].nombre = NULL;
        }
}


// Introducir datos en el array HASH
void ManipularHASH (elemento *a, int n_elementos)
{
        char nom[81];
        int i;
        elemento x;


        i = 0;
        // introducir datos
        printf("\n\t Introducir datos. Finalizar con matricula = 0 \n\n");
        printf ("\n\t ALUMNO No %d", i + 1);

        printf("\n\t Matricula: ");
        scanf("%u", &x.matricula);

        fflush(stdin);

        while (x.matricula != 0)
        {
                printf ("\t Nombre: ");
                gets (nom);

                // Asignar espacio para nombre
                x.nombre = (char *) malloc (strlen (nom) + 1);

                if (!x.nombre)
                        error ();
```

```
                strcpy (x.nombre, nom);

                hash (a, n_elementos, x);      // Llamada a la función hash

                i++;
                printf ("\n\t ALUMNO No %d", i + 1);
                printf("\n\t Matricula: ");
                scanf("%u", &x.matricula);
                fflush(stdin);

                free (x.nombre);
        } // fin del while

} // Fin de ManipularHASH()

// Escribir el array HASH de n_elementos al disco
void GrabarArrayHASH (elemento *a, int n_elementos, FILE *pf)
{
        int i, j;
        int tam_nombre;

        pf = fopen ("Array_HASH", "wb");

        if (pf == NULL)
        {
                perror ("No se pudo abrir el fichero");
                exit (0);
        }

        rewind (pf);

        for (i = 0; i < n_elementos; i++)
        {
                /* Si el registro que se va a escribir en el disco duro, no está vacío,
                   entonces, a continuación escribimos el nombre con la función
                   fputs. Esto se hace así, ya que al ser el nombre un miembro que
                   es un puntero, la función fwrite escribe solamente la dirección
                   de comienzo de la cadena de caracteres que representa al nombre, y
                   no el nombre en si. Para los registros vacíos, NO se escribirá la
                   cadena de caracteres a continuación del registro.
                */

                /* Verificamos si el array está vacío */
                if(a[i].matricula != 0)
                {
                        /* Escribir en el disco un registro con datos validos */
                        fwrite (&a[i], tam_registro, 1, pf);

                        tam_nombre = strlen (a[i].nombre) + 1;
                        putw (tam_nombre, pf);
```

```
                        for (j = 0; j < tam_nombre; j++)
                                fputc (a[i].nombre[j], pf);

                } // cierre de if (a[i].matricula != 0)

        } // cierre del for (i = 0; i < n_elementos; i++)

        fclose (pf);

}
```

/* principal.cpp */

```
/****************** Arrays Hash ***************/
/* Crear un array hash de una determinada longitud L que permita almacenar los datos:
   número de matricula y nombre de los alumnos matriculados en una cierta Universidad,
   utilizando el método hash abierto y la función de acceso mitad del cuadrado.*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <crtdbg.h>
#include "globales.h"


void main ()
{
        elemento *a;            // dirección de comienzo del array
        int n_elementos;        // Número de elementos del array
        int i, j;
        FILE *pf, *pf1;         // Punteros a ficheros
        elemento registro;      // tipo elemento
        // char buffer[N];      // almacena el nombre
        int long_nombre;        // longitud del nombre

        // Averiguar si el array hash ya existe en el disco duro....
        pf =  fopen ("Array_HASH", "rb");
        pf1 = fopen ("cantidad", "rb");

        if (pf != NULL)         // ¿Existe el fichero?
        {
                // El fichero ya existe, reconstruir la lista en memoria principal
                printf ("\n\t !!!ATENCION!!! El fichero ya existe.");
                printf ("\n\t Trayendo datos del disco....");

                // Traer la cantidad de elementos desde el disco
                n_elementos = getw (pf1);
```

```
        // Crear el array dinámico "a"
        a = (elemento *) malloc (n_elementos * sizeof (elemento));
        if(!a)
                error ();

        InicializarHASH (a, n_elementos);

        // Reconstruir el array HASH
        i = 0;
        rewind (pf);
        fread (&registro, tam_registro, 1, pf);

        while (!feof (pf) && !ferror (pf))
        {
                // Obtenemos la longitud del nombre
                long_nombre = getw (pf);

                registro.nombre = (char *) malloc (long_nombre * sizeof (char));

                if (registro.nombre == NULL)
                        error ();

                // Copiamos caracter a caracter el nombre
                for (j = 0; j < long_nombre; j++)
                        registro.nombre[j] = fgetc (pf);

                // Una vez que ya tengo el registro creado en MP, llamar a la
                // funcion HASH para ubicarlo en el array

                hash (a, n_elementos, registro);       // Llamada a la función hash

                free (registro.nombre);

                fread (&registro, tam_registro, 1, pf);
                i++;
        } // cierre del while (!feof (pf) && !ferror (pf))

        printf ("\n\t Datos traidos del disco....\n\n");

        // Imprimimos el array traido del disco
        printf ("\n\t El array traido del disco es....\n\n");
        ImprimirHASH (a, n_elementos);

        // Trabajar con el array HASH creado
        ManipularHASH (a, n_elementos);

        fclose (pf);      // Cerrar el fichero para luego grabar en el disco
        fclose (pf1);

} // cierre del if (pf != NULL)
```

```
	//-------------------------------------------------------
	else
	{
		// El fichero no existe, crear el array
		printf ("\n\t EL FICHERO NO EXISTIA Y SE HA CREADO...");

		printf ("\n\t CONSTRUCCION DE UN ARRAY HASH CON FUNCION DE
				ACCESO MITAD DEL CUADRADO...");
		printf("\n\t Numero de elementos: ");
		scanf("%d", &n_elementos);

		// Crear el array dinámico "a"
		a = (elemento *) malloc (n_elementos * sizeof (elemento));

		if (!a)
			error ();

		InicializarHASH (a, n_elementos);

		// Trabajar con el array HASH
		ManipularHASH (a, n_elementos);

		// 1. Escribir la cantidad de elementos del array al disco
		pf1 = fopen ("cantidad", "wb");
		putw (n_elementos, pf1);
		fclose (pf1);
	} // cierre del else

	// Imprimir el arreglo hash creado
	printf ("\n\t El contenido del array es....");
	ImprimirHASH (a, n_elementos);

	printf ("\n\t Escribiendo el contenido del array al disco...\n");
	// Escribir el array HASH al disco
	GrabarArrayHASH (a, n_elementos, pf);

	// Liberar la memoria asignada para el array a
	// Liberar primero la memoria asignada a los nombres
	for (i = 0; i < n_elementos; i++)
	{
		if (a[i].nombre != NULL)
			free (a[i].nombre);
	}

	free (a);
	if (_CrtDumpMemoryLeaks())
		printf ("\n\t Se encontraron LAGUNAS DE MEMORIA...");
}
```

SALIDA DEL PROGRAMA

En la Ilustración 6-95, se muestra la salida de la primera ejecución de programa. Igual que antes, las entradas de datos por parte del usuario, están mostradas en negritas, cursiva y subrayadas.

```
      EL FICHERO NO EXISTIA Y SE HA CREADO...
      CONSTRUCCION DE UN ARRAY HASH CON FUNCION DE ACCESO MITAD DEL CUADRADO
..
      Numero de elementos: 20

      Introducir datos. Finalizar con matricula = 0

      ALUMNO No 1
      Matricula: 200
      Nombre: Juan

      ALUMNO No 2
      Matricula: 0

      El contenido del array es....
      ELEMENTOS DEL ARRAY HASH...
     ---------------------------------

      ELEMENTO 0 DEL ARRAY....
      Matricula: 0
      Nombre: (null)

      ELEMENTO 1 DEL ARRAY....
      Matricula: 0
      Nombre: (null)

      ELEMENTO 2 DEL ARRAY....
      Matricula: 200
      Nombre: Juan

      ELEMENTO 3 DEL ARRAY....
      Matricula: 0
      Nombre: (null)

      ELEMENTO 4 DEL ARRAY....
      Matricula: 0
      Nombre: (null)

      ELEMENTO 5 DEL ARRAY....
      Matricula: 0
      Nombre: (null)

      ELEMENTO 6 DEL ARRAY....
      Matricula: 0
      Nombre: (null)

      ELEMENTO 7 DEL ARRAY....
      Matricula: 0
```

```
        Nombre: (null)

        ELEMENTO 8 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 9 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 10 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 11 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 12 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 13 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 14 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 15 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 16 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 17 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 18 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 19 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        Escribiendo el contenido del array al disco...
Press any key to continue
```

Ilustración 6-95: Primera salida del ejemplo HASH con función de acceso MITAD DEL CUADRADO

En la Ilustración 6-96, se muestra la salida de la primera ejecución de programa. Igual que antes, las entradas de datos por parte del usuario, están mostradas en negritas, cursiva y subrayadas.

```
!!!ATENCION!!! El fichero ya existe.
Trayendo datos del disco....
Datos traidos del disco....

El array traido del disco es....

ELEMENTOS DEL ARRAY HASH...
--------------------------------------------------

ELEMENTO 0 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 1 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 2 DEL ARRAY....
Matricula: 200
Nombre: Juan

ELEMENTO 3 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 4 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 5 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 6 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 7 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 8 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 9 DEL ARRAY....
Matricula: 0
Nombre: (null)
ELEMENTO 10 DEL ARRAY....
Matricula: 0
```

```
Nombre: (null)

ELEMENTO 11 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 12 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 13 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 14 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 15 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 16 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 17 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 18 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 19 DEL ARRAY....
Matricula: 0
Nombre: (null)

Introducir datos. Finalizar con matricula = 0

ALUMNO No 1
Matricula: 280
Nombre: Veronica

ALUMNO No 2
Matricula: 0

El contenido del array es....

ELEMENTOS DEL ARRAY HASH...
--------------------------------------------------

ELEMENTO 0 DEL ARRAY....
Matricula: 0
Nombre: (null)
```

```
ELEMENTO 1 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 2 DEL ARRAY....
Matricula: 200
Nombre: Juan

ELEMENTO 3 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 4 DEL ARRAY....
Matricula: 280
Nombre: Veronica

ELEMENTO 5 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 6 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 7 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 8 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 9 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 10 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 11 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 12 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 13 DEL ARRAY....
Matricula: 0
Nombre: (null)

ELEMENTO 14 DEL ARRAY....
Matricula: 0
Nombre: (null)
```

```
        ELEMENTO 15 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 16 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 17 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 18 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        ELEMENTO 19 DEL ARRAY....
        Matricula: 0
        Nombre: (null)

        Escribiendo el contenido del array al disco...
Press any key to continue
```

Ilustración 6-96: Segunda salida del ejemplo HASH con función de acceso MITAD DEL CUADRADO

EXPLICACIÓN DE LA CODIFICACIÓN DEL PROGRAMA

Básicamente, este programa tiene la misma estructura y funcionalidad que el programa anterior. La única diferencia aquí es que la función de acceso que se implementa, es la de la mitad del cuadrado.

La estructura de datos que se ha usado para almcenar los datos de los estudiantes, ha sido exactamente la misma.

La variación ha consistido en la función MitadDelCuadrado, la cual se ha usado para obtener el índice dentro del arreglo, dado un valor de clave.

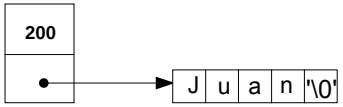
Explicaremos gráficamente lo que ha sucedido dentro de dicha función, al ejecutar el programa con los valores introducidos y los resultados obtenidos en la Ilustración 6-95 y Ilustración 6-96; específicamente para el alumno Juan con clave 200. En la Ilustración 6-97, se muestra lo expuesto anteriormente.

En la Ilustración 6-98, se muestra la inserción, en una segunda ejecución del programa, del alumno Verónica, con matrícula 280.

Cálculo del valor del índice para un valor de clave 200 y una longitud del array de 20

L = 20 (Longitud del array)

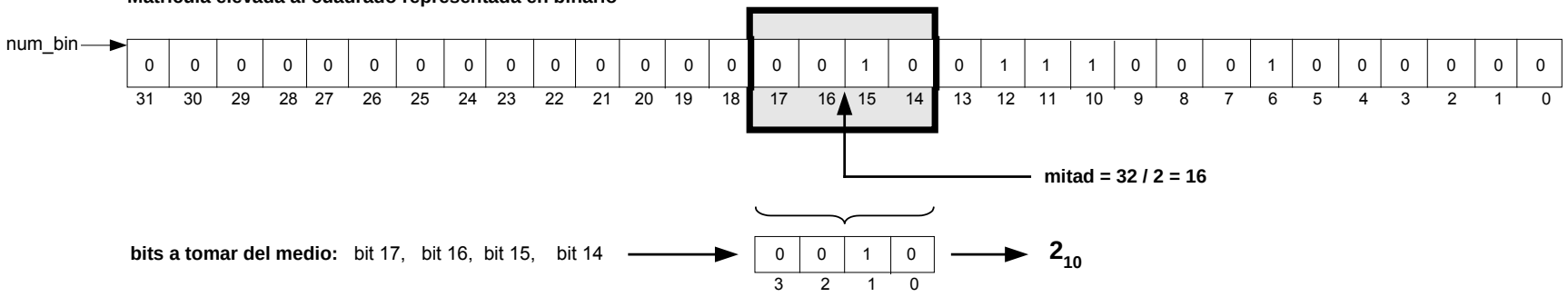
Matricula = 200 (Clave)



$$n = \frac{\log_{10} 20}{\log_{10} 2} = 4$$ → **n**: bits a tomar del medio del número binario que representa la clave elevada al cuadrado

Matricula^2 = 40000

Matrícula elevada al cuadrado representada en binario



Entonces, la posición (índice) donde será insertado el elemento, será la posición 2

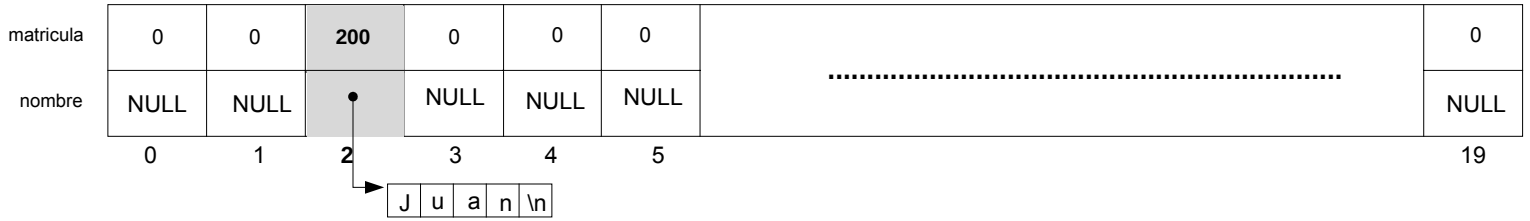
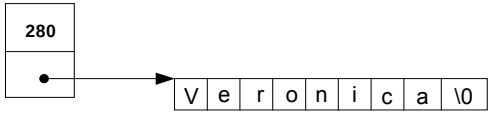


Ilustración 6-97: Cálculo del índice en un array HASH con la función de acceso MITAD DEL CUADRADO

Cálculo del valor del índice para un valor de clave 280 y una longitud del array de 20

L = 20 (Longitud del array)

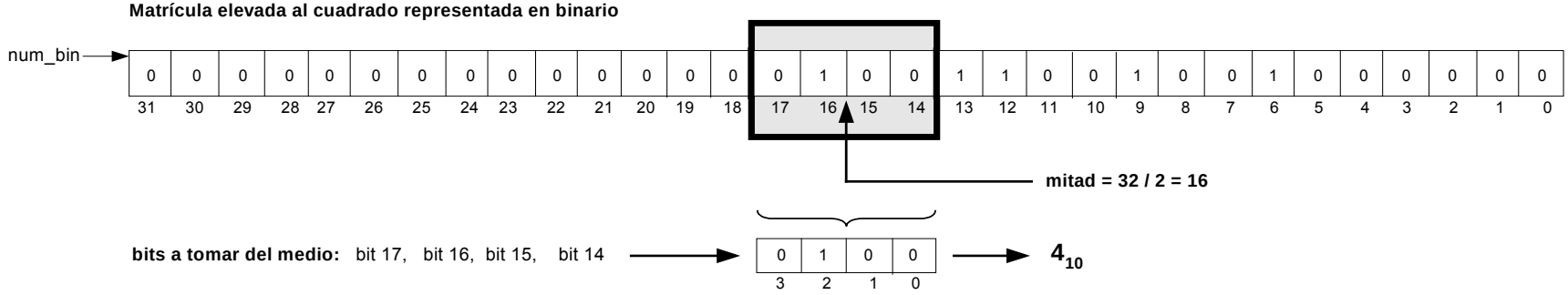
Matricula = 280 (Clave)



$$n = \frac{\log_{10} 20}{\log_{10} 2} = 4$$ → **n**: bits a tomar del medio del número binario que representa la clave elevada al cuadrado

Matricula^2 = 78400

Matrícula elevada al cuadrado representada en binario



Entonces, la posición (índice) donde será insertado el elemento, será la posición 4

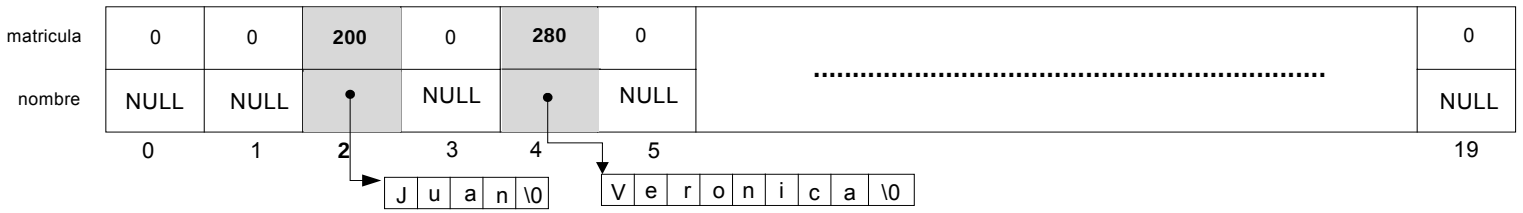


Ilustración 6-98: Cálculo del índice en un array HASH con la función de acceso MITAD DEL CUADRADO

ALGORITMO DE LAS PRINCIPALES FUNCIONES

Función Mitad del Cuadrado

integer Function MitadDelCuadrado (int n_elementos, int matricula)

// Declaración de Variables y Constantes
constante NUM_MAXIMO_BITS ← 32
var n: integer;
var matri_cuad: unsigned long;
num_bin: array of integer [NUM_MAXIMO_BITS]
num_bin_mitad: pointer to integer;
var decimal: integer;

// Pasos a seguir
// 1. Calcular los n bits del medio a tomar del número binario que representa la
// matrícula al cuadrado
n ← $\log_{10}$ (n_elementos) / $\log_{10}$ (2)

// 2. Elevar clave al cuadrado
matri_cuad ← (matricula ^ 2)

// 3. Convertir matri_cuad a binario puro, el resultado se almacenará en num_bin
ConvertirA_BinPuro (matri_cuad, num_bin)

// 4. Obtener el indice mitad del arreglo num_bin
mitad ← (NUM_MAXIMO_BITS / 2)

// 5. Asignar memoria para almacenar el binario de la mitad del numero
num_bin_mitad ← AsignarMemoria (n)
Si la asignación no ha sido posible, entonces
    Generar mensaje de error y salir
FinSi

// 6. Calcular la posición inicial desde donde se empezarán a tomar los bits a
// copiar
aux ← mitad - (n / 2)

// 7. Copiar los bits desde la posición dada por aux hasta aux + n
Para i ← 0 hasta n, hacer
    num_bin_mitad[i] ← num_bin[aux]
    aux++;
FinPara

// 8. Convertir a Decimal el numero binario obtenido en el paso 7
decimal ← ConvertirBP_A_Decimal (num_bin_mitad, n)

// 9. Liberar memoria para el puntero num_bin_mitad
LiberarMemoria (num_bin_mitad)

// 10. Retornar el valor en decimal
Retornar (decimal)

Fin Function

Nota: Para el cálculo de la cantidad de bits a tomar del medio, se han seguido las siguientes operaciones.

$$2^n \leq L$$

$$\log_{10} 2^n \leq \log_{10} L$$

$$n * (\log_{10} 2) \leq \log_{10} L$$

$$n \leq \frac{\log_{10} L}{(\log_{10} 2)},$$

donde L: Longitud del array HASH

7. Conclusiones

Al finalizar este trabajo, podemos concluir lo siguiente:

- Se ha elaborado con éxito la codificación de programas en Lenguaje C, debidamente documentados, relacionados con las técnicas de Organización de Archivos.

- Se han desarrollado una diversidad de ejercicios resueltos con ilustraciones explicativas, respecto al capítulo de Organización de Archivos.

- Se ha podido demostrar que los aspectos teóricos ofrecidos por los docentes del área de Base de Datos, específicamente los relacionados con el capítulo Organización de Archivos, pueden ser llevados a la práctica, mediante el uso de un Lenguaje de Programación y el conocimiento preciso de dichos aspectos teóricos.

8. Recomendaciones

- Que se sigan desarrollando implementaciones acerca de los restantes temas de Base de Datos, por ejemplo: transacciones, procesamiento de consultas u otros.

- Que los programas que hemos presentado en este documento, puedan ser elaborados por estudiantes, utilizando las funciones gráficas del Lenguaje C estándar.

9. Bibliografía

- Cairó Osvaldo, Guardati Silvia. *Estructura de Datos*. Segunda Edición. McGraw-Hill 2004.

- Loomis S Mary. *Estructura de datos y organización de archivos*. Segunda Edición. Prentice – Hall. 1991

- Korth, Silberschatz y Sudarshan. *Fundamentos de Base de Datos*. Cuarta Edición. McGraw-Hill. 2003

- Ceballos F. Javier. *Enciclopedia del Lenguaje C*. Alfaomega – Rama. 1997

Otras Referencias

Transparencias

- Espinoza Ernesto, Espinoza Ricardo. *Transparencias de la asignatura Base de Datos II*. 2002 - 2003

Internet:

- http://www.algoritmia.net
- http://www.udlap.mx/~sainzmar/is211/b.htm
- http://www.lawebdelprogramador.com
- http://www.cconclase.net
- http://www.elrincondelc.com