Estructura de datos.

Objetivo

Otorgar al participante el conocimiento, la habilidad y la aptitud para: comprender y manejar las representaciones más utilizadas para el procesamiento de información en sistemas de computación. Conocer los diferentes métodos de búsqueda y ordenamiento y seleccionar y aplicar el algoritmo más adecuado para la solución a problemas de ingeniería.

Temario General

1. Repaso de apuntadores.
2. Estructuras.
3. Pilas.
4. Colas.
5. Listas.
6. Árboles.
7. Ordenamientos.
8. Búsquedas.
9. Archivos.

Bibliografía

Tenembaum, A. N. Augenstein, J. J. **Estructuras de Datos en C.** Prentice-Hall. México. 1991.

Ullman, J., Aho, A. y Hopcroft, J. **Estructuras de Datos y Algoritmos.** Addison-Wesley. México. 1988.

Joyanes, L. **Fundamentos de Programación. Algoritmos y estructura de datos.** McGraw-Hill. México. 1990.

Cairó, Osvaldo. **Estructuras de datos.** McGraw-Hill. México. 1993.

Apuntadores.

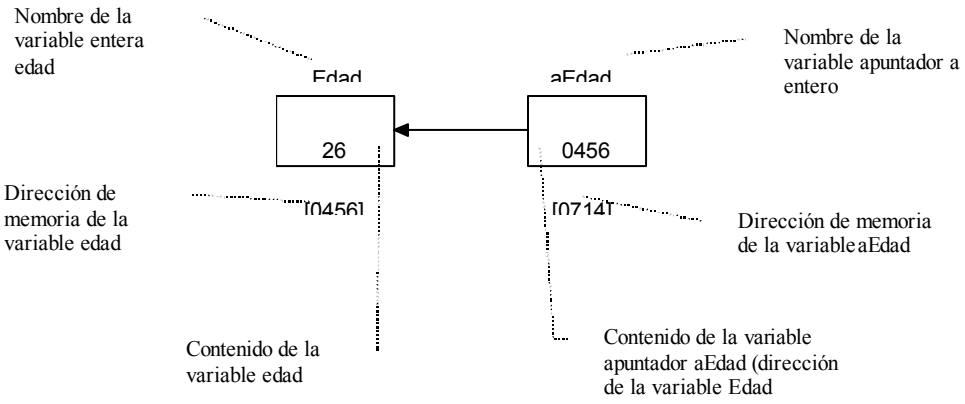
Un apuntador es una variable que contiene una dirección de memoria. Esta dirección puede ser la posición de otra variable en la memoria.

Por ejemplo:

int edad, *aEdad;

Edad=26;

AEdad=&Edad;



En el ejemplo tenemos dos variables una de tipo entero llamada Edad y otra de tipo apuntador a entero llamada aEdad. Posteriormente Edad toma el valor de 26 y aEdad toma como valor la dirección de la variable Edad (0456). Podemos decir entonces que aEdad apunta a Edad, y a través de aEdad puedo modificar el valor de la variable Edad.

La declaración de un apuntador como ya vimos en el ejemplo se hace:

tipo *nombre_var;

Se agrega * como prefijo al declarar una variable de tipo apuntador. Tipo define el tipo de dato al que va a hacer referencia el apuntador.

Operadores de apuntador.

& Operador de dirección o referencia. Devuelve la dirección de memoria de la variable.

* Operador de indirección o "desreferencia"[1]. Devuelve el valor situado en la dirección del operando. Se dice que da acceso a la variable que señala el apuntador.

Ejemplos. Supongamos un apuntador p y dos variables c y d de tipo char.

Char *p, c, d;

p=&c; Se asigna la dirección de c a la variable apuntador p

d=*p; Asigna el contenido de c (al que apunta p) a la variable d

Un pequeño programa de ejemplo: Inicializa una variable entera i con el valor de 100, posteriormente se asigna la dirección de i al apuntador pi. Después la variable *val* recibe el contenido de lo apuntado por pi (es decir 100) y finalmente se despliega el contenido de val.

```
#include<stdio.h>
void main(){
        int *pi, i, val;
        i=100;
        pi=&i;
        val=*pi;
        printf("%d", val);
}
```

[1] Desreferencia como muchas otras palabras utilizadas en el ámbito de la computación no existe realmente en el español, pero es utilizado por algunos autores o por la traducción literal de documentos en inglés.

Operadores aritméticos para apuntadores.

Las únicas operaciones que se pueden realizar con variables de apuntador son la suma y la resta, de manera que los operadores válidos son:

+ suma

– resta

++ incremento

-- decremento

La aritmética de operadores no suma las direcciones de memoria, sino elementos. Esto quiere decir que si yo incremento en uno a una variable de apuntador a entero no se incrementara un byte, sino dos bytes porque es el espacio que ocupa un entero en memoria.[2]

Más ejemplos con apuntadores:

```
int x=10, y=2, z[14], *p;
```

p=&x; p apunta ahora a la variable x

y=*p; y contiene ahora el valor de 10

*p=0; x es asignada con un valor de cero a través del apuntador p.

p=&z[2]; p apunta ahora a z[2]

*p=*p+2; incrementa en dos lo apuntado por p.

++*p incrementa en uno lo apuntado por p.

(*p)++ incrementa en uno lo apuntado por p. los paréntesis son necesarios para que no incremente a p.

[2] Es lógico que así funcione ya que lo práctico aquí es apuntar al siguiente entero (por ejemplo recorrido de un arreglo).

Estructuras

Una **estructura** es un conjunto de variables que se citan y manejan con el mismo nombre y que permite además la utilización individual de sus elementos.

La **estructura** en C es muy **similar**, en concepto, al **registro** en PASCAL, FORTRAN, etc.

Una **definición de estructura** forma una **plantilla** o **patrón** que puede utilizarse para crear variables de estructura que, con diferentes nombres, se ajusten a esa plantilla.

El formato de definición de una variable estructura es el siguiente:

```
struct NombreEstructura
   {
        TipoVariable1 NombreVariable1 ;
        TipoVariable2 NombreVariable2 ;
        TipoVariable3 NombreVariable3 ;
                      M
        TipoVariableN NombreVariableN ;
   }
NombVarEstru1, NombVarEstru2, K, NombVarEstruN ;
```

o bien:

```
struct NombreEstructura
   {
        TipoVariable1 NombreVariable1 ;
        TipoVariable2 NombreVariable2 ;
        TipoVariable3 NombreVariable3 ;
                      M
        TipoVariableN NombreVariableN ;
   } ;
```

struct NombreEstructura NombVarEstru1, NombVarEstru2, **K**, NombVarEstruN ;

Si solamente se necesita una variable estructura, por ejemplo NombVarEstru1, no es necesario poner el nombre de la estructura (NombreEstructura) y quedaría el siguiente formato:

```
struct
   {
        TipoVariable1 NombreVariable1 ;
        TipoVariable2 NombreVariable2 ;
        TipoVariable3 NombreVariable3 ;
                      M
        TipoVariableN NombreVariableN ;
   } NombVarEstru1 ;
```

Ejemplo:

```
struct
{
        char apellidos[35];
        char nombre[25];
        char direccion[40];
        char telefono[7];
        float saldo;
        float debe;
}
cuenta;
```

crea una variable estructura de nombre **cuenta** que podrá ser utilizada en un conjunto, como **una sola variable**, citando el nombre de **cuenta**.

¿Y cómo se utilizan los elementos individuales de la estructura?

El lenguaje C permite el **manejo individual** de los **elementos** de una variable estructura por medio del operador **punto** (**.**).

Para referenciar los elementos individuales de la estructura anterior podemos usar las siguientes representaciones:

```
cuenta.apellidos                  /* Referencía al arreglo apellidos */
cuenta.apellidos[0]               /* Referencia al primer carácter de apellidos */
cuenta.saldo                      /* Referencia al elemento saldo */
printf ("%s", cuenta.direccion);  /* Imprime el elemento dirección */
```

Ejemplos: dirección, teléfono, fecha, hora, nombre, descripción física, libro, alumno, etc.

cuenta	apellidos[0]	´G´
		M
	apellidos[34]	´ ´
	nombre[0]	´D´
		M
	nombre[24]	´ ´
	direccion[0]	´C´
		M
	direccion[39]	´ ´
	telefono[0]	´1´
		M
	telefono[6]	´2´
	saldo	1000
	debe	500

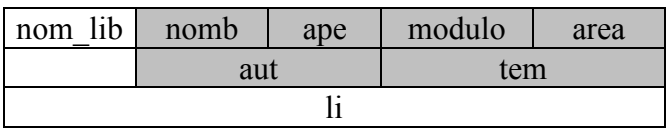
Estructuras Anidadas

Una estructura puede **anidar** otras estructuras, previamente definidas, como se demuestra en el siguiente ejemplo en el que las estructuras se definen a nivel **global**.

Para mencionar un elemento de la estructura se indican, ordenadamente, el nombre de la estructura principal, el nombre de la estructura **anidada** y el nombre del elemento de la estructura, todos ellos separados por el **operador punto (.)**

```
#include <stdio.h>
#include <conio.h>
struct autor                /* Declara GLOBALMENTE autor */{
        char nomb[25];
        char ape[35];
};
struct tema                 /* Declara GLOBALMENTE tema */{
        char modulo[4];
        char area[20];
};
 /* Estructura "libros" que anida las estructuras "autor"y "tema".
    La declaración es también GLOBAL. */
struct libros{
        char nom_lib[70];
        struct autor aut;
        struct tema tem;
};
void main(){
        struct libros li;                   /* Declara li del tipo libros */
        clrscr();                           /* Borra la pantalla */
        puts("Titulo del libro");           /* Imprime en pantalla */
        gets(li.nom_lib);                   /* Lee datos del teclado */
        puts("Apellidos del autor");
        gets(li.aut.ape);
        puts("Nombre del autor");
        gets(li.aut.nomb);
        puts("Modulo:");
        gets(li.tem.modulo);
        puts("Area de conocimiento");
        gets(li.tem.area);
}
```

El anidamiento de estructuras gráficamente se visualizaría así:



Arreglos de estructuras

De igual forma que las variables, también una estructura puede formar parte de un arreglo. Para ello, **primero** debemos **definir la estructura** y a continuación declarar una variable arreglo de dicho tipo.

El formato general es el siguiente:

```
struct NombreEstructura
   {
        TipoVariable1 NombreVariable1 ;
                      M
        TipoVariableN NombreVariableN ;
    } ;
```

struct NombreEstructura VariableArreglo[NumElementos];

Se utiliza el operador punto para citar elementos individuales de la variable estructura e incluso elementos individuales del arreglo utilizando el subíndice correspondiente.

Como cualquier otra variable, una estructura puede tener un almacenamiento **global**, **local** o **externo**.

Ejemplo en C:

```
struct libreria                  /* Comienzo de definición del "patrón". */
{
        char titulo[100];
        char autor[75];
        unsigned long precio;
};                               /* Fin de definición del "patrón. */

void main()                      /* Inicio de la función main() */
{
        /* Declara la variable libros[NumElementos] de "tipo" librería */

        struct libreria libros[10];

        int n, i;
        Λ
        Λ

    /* Se captura la información de n libros */

        printf ("Cuantos libros: \n");
        scanf ("%d", &n);                        fflush(stdin);
        for (i=0; i<n; i++)
                {
```

```
            printf ("\n\t\tLibro %d:\n", i+1);
            puts("Titulo : ");
            gets (libros[i].titulo);                    fflush(stdin);
            puts("Autor : ");
            gets (libros[i].autor);                     fflush(stdin);
            puts("Precio : ");
            scanf ("%ul", &libros[i].precio);           fflush(stdin);
        } /* del for */
}                               /* Fin de la funcion main() y del programa */
```

Se declara **globalmente** la estructura "librería" y dentro de la función main() se define la variable **libros[1000]** del **tipo** librería.Gráficamente el arreglo de estructuras tipo **librería** se visualizaría así:

titulo	autor	precio	titulo	autor	precio	Λ	titulo	autor	precio
libros[0]			libros[1]				libros[9]		

Otra forma de declarar una estructura es emplear la definición **typedef** en C. Por ejemplo:

```
typedef struct  {
        char titulo[100];
        char autor[75];
        unsigned long precio;
    } LIBRERIA;
```

dice que el identificador LIBRERÍA es sinónimo del especificador de estructura anterior dondequiera que ocurra LIBRERÍA. Entonces, podemos declarar

LIBRERÍA libro_comp, libro_mat, libros[10];

El estándar ANSI de C permite la asignación de estructuras del mismo tipo. Por ejemplo el enunciado **libro_comp = libro_mat**, es válido y equivalente a

```
libro_comp.titulo   =   libro_mat.titulo;
libro_comp.autor    =   libro_mat.autor;
libro_comp.precio   =   libro_mat.precio;
```

y estas líneas, a su vez, son equivalentes a

```
for (i=0; i<100; i++)
        libro_comp.titulo[i] = libro_mat.titulo[i];
for (i=0; i<75; i++)
        libro_comp.autor[i] = libro_mat.autor[i];
libro_comp.precio = libro_mat.precio;
```

Muchos compiladores, basados en el lenguaje C original definido por Kernighan y Ritchie, **no permiten asignación de estructuras**. Por tanto, sería necesario **asignar explícitamente cada elemento** de una estructura a otra.

Tampoco es posible comparar la igual de dos estructuras en una sola operación en C. Es decir, **libro_comp == libro_mat** no está definido.

Estructuras como parámetros

Para pasar una estructura a una función, debemos pasar su **dirección** a la función y referirse a la estructura mediante un **apuntador**, es decir, paso de parámetros por **referencia**.

Para referirse a un **elemento** de una estructura **pasada** como parámetro por **referencia** se utiliza el **operador -> (guión medio y un símbolo de mayor).**

Operadores de Seleccion.

Existen **dos operadores de selección** para tener **acceso** a una estructura: **.** y **->**

. Selector de miembro **directo**
-> Selector de miembro **indirecto** o **apuntador**

Los operadores de selección **.** y **->** se usan para **accesar** los **miembros** de una **estructura** y de una **unión**.

Suponga que el objeto **s** es de tipo estructura **S** y **sptr** es un apuntador a **S**. Entonces, si **m** es un identificador de miembro de tipo **M** declarado en **S**, estas expresiones:

```
s.m
sptr->m
```

son de tipo **M**, y ambas representan el objeto miembro **m** en **s**.

La expresión

```
sptr->m
```

es un sinónimo conveniente para **(*sptr).m**

Selector de miembro **directo** (**.**)

Sintaxis:
expresión_postfija. identificador

- La expresión postfija debe ser de tipo unión o estructura.
- El identificador debe ser el mismo nombre de un miembro de ese mismo tipo de estructura o unión.

Operador de miembro **indirecto** (**->**)

Sintaxis:
expresión_postfija -> identificador

- La expresión postfija debe ser de tipo apuntador a estructura o apuntador a unión.
- El identificador debe ser el mismo nombre de un miembro de ese mismo tipo de estructura o unión.

La expresión designa a un miembro objeto de una estructura o unión. El valor de la expresión es el valor del miembro seleccionado.

Por ejemplo,

```
struct mi_estructura {
   int i;
   char cadena[21];
   double d;
} s, *sptr=&s;

...

s.i = 3;                  // asigna al miembro i de mi_estructura s
sptr->d = 1.23;       // asigna al miembro d de mi_estructura s
```

UNIONES

Una **unión** es una zona de memoria utilizable por variables de **tipos** diferentes.

La definición de una unión se realiza de forma similar a la de una estructura:

```
union NombreUnion
{
   TipoVar1 NombreVar1 ;
   TipoVar2 NombreVar2 ;
        TipoVar3 NombreVar3 ;
                    M
        TipoVarN NombreVarN ;
};
union NombreUnion Varia1Union, Varia2Union, . . . , VariaNUnion;
```

Cuando se declara una unión y sus variables asociadas, el compilador reserva el espacio necesario para el **mayor** de los elementos comprendidos **dentro** de la unión, para cada **una** de las variables **definidas** del tipo de la unión.

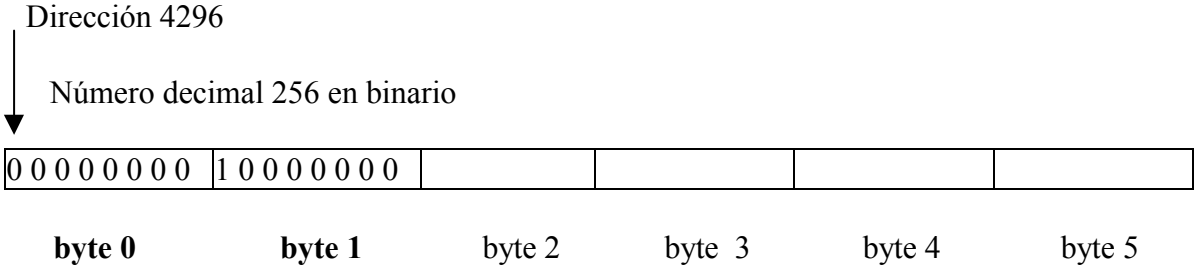
La asignación de valores a las variables asociadas a la unión se hace, al igual que en las estructuras, utilizando el operador punto y el operador flecha si se trata de manejar apuntadores.

Todos los elementos integrados dentro de la unión comienzan en la misma dirección de memoria, por lo cual pueden superponerse.

Salida en pantalla del Ejemplo en C:

```
El espacio reservado para CADA variable ASOCIADA a la UNION es de 6 bytes.

Las Direcciones de memoria de los elementos INTEGRADOS en la UNION son:

&t.ncar[0] = 4296
&t.entero = 4296

El byte "0" de 256 es 0
El byte "1" de 265 es 1
```

```
#include <stdio.h>

void main()
{

union tipos
    {
        unsigned char car[6];
        int entero;
    };

union tipos t;

t.entero = 256;

printf("\n\nEl espacio reservado para CADA variable ASOCIADA a la union es");
printf("de %d bytes. \n\n", sizeof(union tipos));
printf("Las direcciones de memoria de los elementos INTEGRADOS en la ");
printf(" UNION son: \n\n");
printf("&t.ncar[0] = %u",  &t.car[0]);
printf("\n&t.entero = %u\n\n", &t.entero);
printf("El byte \"0\" de %d es %d\n", t.entero, t.car[0]);
printf("El byte \"1\" de %d es %u\n", t.entero, t.car[1]);
}
```



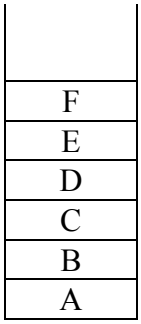
PILA

Una **pila** es un conjunto ordenado de elementos en el cual se pueden agregar y eliminar elementos en un extremo, que es llamado el **tope** de la pila.

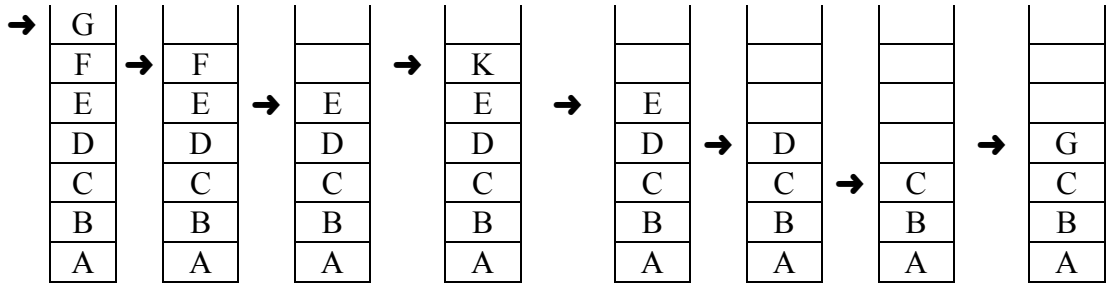
La definición de la pila considera la **inserción** y **eliminación** de elementos, por lo que una pila es un **objeto dinámico** en constante cambio.

La definición especifica que un solo extremo de la pila se designa como el tope.

Pueden colocarse nuevos elementos en el tope de la pila o se pueden quitar elementos.



Una pila con 6 elementos



pop (s) pop (s) push (s, K) pop (s) pop (s) pop (s) push (s, G)

La característica más importante de una pila es **que el último elemento insertado en ella es el primero en suprimirse**. Por esta razón, en ocasiones una pila se denomina una lista "último en entrar, primero en salir" o LIFO (**l**ast **i**n, **f**irst **o**ut).

En la pila **no se conserva un registro de los elementos intermedios** que han estado en ella, si se desea conservar, debe llevarse en otra parte.

Operaciones primitivas

Los dos cambios que pueden hacerse en una pila reciben nombres especiales.

Cuando se añade un elemento a una pila, se **agrega** a la pila y cuando se quita un elemento, se **remueve** de la pila.

Dada una pila *s* y un elemento *i*, ejecutar la operación ***push (s,i)*** agrega el elemento *i* al tope de la pila *s*.

De modo contrario, la operación ***pop (s)*** remueve el elemento superior y lo regresa como su valor de función.

La operación de asignación

i = *pop (s)*;

resuelve el elemento del tope de *s* y le asigna su valor a *i*.

No hay un **límite** de la cantidad de elementos que pueden conservarse en una pila. Sin embargo, si una pila contiene un solo elemento y éste se remueve de ella, la pila resultante no contiene elementos y se llama **pila vacía**.

Aunque la operación *push* es aplicable a cualquier pila -si la memoria lo permite-, la operación *pop* no puede aplicarse a la pila vacía porque esta pila no tiene elementos para remover.

Antes de aplicar el operador *pop* a una pila **debemos asegurarnos de que la pila no está vacía**.

La operación *empty (s)* determina si una pila *s* está o no vacía. Si la pila está vacía, *empty (s),* retorna el valor **TRUE**; de otra forma retorna el valor **FALSE**.

Otra operación que puede ejecutarse sobre una pila es determinar cuál es el elemento superior sin quitarlo. Esta operación se escribe *stacktop (s)* y retorna el elemento superior de la pila *s*.

En realidad la operación *stacktop (s)* no es nueva, porque consta de las operaciones de remover (*pop*) y agregar (*push*).

i = *stacktop (s)*;

es equivalente a

i = *pop (s)*;
push (s,i);

Igual que la operación *pop*, no se define *stacktop* para una pila vacía. El resultado de un intento no válido por remover o acceder un elemento de una pila vacía se llama **subdesbordamiento**. **Éste se evita segurándose de que *empty (s)* sea falso antes de intentar la operación *pop (s)* o *stacktop (s)*.**

Una representación en C.

Hay varias formas de representar una pila en C. Por el momento, consideraremos la más simple de ellas. Cada una tiene ventajas y desventajas, en términos de la fidelidad con la que refleja el concepto abstracto de una pila y el esfuerzo que deben aportar el programador y la computadora para usarla.

Una pila en C se declara como una **estructura** que contiene **dos objetos**: un **arreglo** para contener los elementos de la pila y un **entero** para indicar la posición del tope de la pila actual dentro del arreglo.

```
#define STACKSIZE 100
struct stack {
        int top;
        int items[STACKSIZE];
};
```

Una vez que se ha hecho esto, de declara una pila real *s* mediante

```
struct stack s;
```

Aquí suponemos que los elementos de la pila *s* contenidos en el arreglo *s.items* son enteros y que la pila en ningún momento contendrá más enteros que STACKSIZE.

El identificador *top* siempre debe declararse como entero, dado que su valor representa la posición dentro del arreglo *items* del elemento en el tope de la pila.

La pila vacía no contiene elementos y, por tanto, se indica mediante ***top*** igual a **–1**. Para inicializar una pila *s* para un estado vacío, se ejecuta al inicio **s.top=-1**:

```
int empty (struct stack *ps)
{
  if (ps->top == -1)
        return (TRUE);
  else
        return (FALSE);
} /* fin de empty */
```

En funciones como *pop* y *push*, las cuales modifican sus argumentos de estructura, al igual que *empty* (que no lo hace) adoptamos la convención de pasar la dirección de la estructura de la pila.

Implementación de la operación *pop*

La posibilidad de un **subdesbordamiento** debe considerarse al implementar la operación *pop*, dado que el usuario puede intentar remover un elemento de una pila vacía. La función *pop* ejecuta las acciones siguientes:

1. Si la pila está vacía, imprime un mensaje de advertencia y detiene la ejecución.
2. Quita el elemento superior de la pila.
3. Retorna este elemento al programa que llama.

```
int pop (struct stack *ps)
{
  if (empty (ps)) {
     printf ("stack underflow");
     exit (1);
  } /* fin del if */
   return (ps->items[ps->top--]);
} /* fin de pop */
```

La llamada a la función *pop* sería así: x = pop (&s);

```
if (!empty (&s))
     x = pop (&s);
else
     /* realiza acciones correctivas */
```

Implementación de la operación *push*

```
void push (struct stack *ps, int x)
{
   if (ps->top == STACKSIZE-1) {
      printf ( "stack overflow");
      exit (1);
   }
   else
      ps->items[++(ps->top)] = x;
   return;
} /* fin de push */
```

La llamada a la función *push* sería: push (&s, x);

Comentarios

Las pilas son una estructura de datos muy usada en la solución de diversos tipos de problemas. Ahora se verán algunos de los casos más representativos de aplicación de pilas:

- Llamadas a subprogramas
- Recursión
- Tratamiento de expresiones artiméticas
- Ordenación

Llamadas a subprogramas

Cuando se tiene un programa que llama a un subprograma, internamente se usan pilas para guardar el estado de las variables del programa en el momento que se hace la llamada. Así, cuando termina la ejecución del subprograma, los valores almacenados en la pila pueden recuperarse para continuar con la ejecución del programa en el punto en el cual fue interrumpido. Además de las variables, debe guardarse la dirección del programa en la que se hizo la llamada, porque es a esa posición a la que regresa el control del proceso.

Recursión

Aplicación de pilas en procesos recursivos.

Tratamiento de expresiones aritméticas

Un problema en computación es poder convertir expresiones en notación infija a su equivalente en notación postfija (o prefija). La ventaja de usar expresiones en notación polaca postfija o prefija radica en que no son necesarios los paréntesis para indicar orden de operación, ya que éste queda establecido por la ubicación de los operadores con respecto a los operandos.

ordenación

Otra aplicación de las pilas puede verse en el método de ordenación rápida.

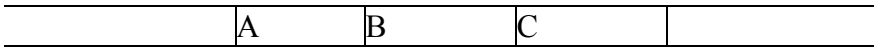
Colas.

Una **cola** es un conjunto ordenado de elementos del que pueden suprimirse elementos de un extremo (llamado la **parte delantera** de la cola) y en el que pueden insertarse elementos del otro extremo (llamado la **parte posterior** de la cola).

El primer elemento insertado en una cola es el primer elemento que se suprime.

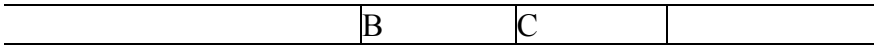
Por esta razón, una cola se denomina una lista **fifo** (el primero en entrar, el primero en salir) que es lo contrario de una pila, la cual es una lista **lifo** (el último en entrar, el primero en salir).

Parte delantera



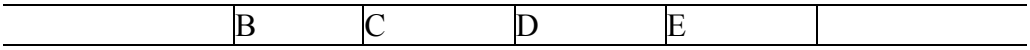
Parte posterior

Parte delantera



Parte posterior

Parte delantera



Parte posterior

Se aplican tres operaciones primitivas a una cola:

1. La operación *insert (q,x)* inserta el elemento *x* en la parte posterior de la cola *q*.

2. La operación *x = remove (q)* suprime el elemento delantero de la cola *q* y establece su contenido en *x*.

3. La operación, *empty (q)* retorna *false* o *true*, dependiendo si la cola contiene elementos o no.

Podría considerarse una operación adicional que indique si la cola se encuentra llena.

La operación *insert* puede ejecutarse siempre, pues no hay un límite -teórico- en la cantidad de elementos que puede contener una cola. Sin embargo, la operación *remove* sólo puede aplicarse si la cola no está vacía; no hay forma de remover un elemento de una cola que no contiene elementos. El resultado de un intento no válido de remover un elemento de una cola vacía se denomina **subdesbordamiento**. La operación *empty* siempre es aplicable.

Implementación de colas en C

Una idea es usar un arreglo para contener los elementos de la cola y emplear dos variables, *front* (parte delantera) y *rear* (parte posterior) para contener las posiciones dentro del arreglo de los elementos primero y último de la cola.

```
#define MAXQUEUE 100
struct queue {
        int items [MAXQUEUE];
        int front, rear;
};
```

Usar un arreglo para que contenga una cola introduce la posibilidad de **desbordamiento**, si la cola llega a ser más grande que el tamaño del arreglo.

La operación *insert (q, x),* podría implementarse mediante la sentencia

```
q.items[++q.rear] = x;
```

y la operación **x = remove (q)** mediante

```
x = q.items[q.front++];
```

Al principio, se establece ***q.rear*** en –1 y ***q.front*** en 0.

La cola está vacía cada vez que **q.rear < q.front**. La cantidad de elementos en la cola en cualquier momento es igual al valor de **q.rear – q.front + 1**.

Bajo este esquema es posible llegar a la absurda situación en donde la cola está llena, aun cuando no se han insertado elementos nuevos.

Una solución es modificar la operación *remove* para que cuando se suprima un elemento, toda la cola cambie al principio del arreglo.

```
x =  q.items[0];
for (i=0; i<q.rear; i++)
        q.items[i] = q.items[i+1];
q.rear--;
```

La cola ya no necesita contener un campo *front*, porque el elemento en la posición 0 del arreglo siempre está en la parte delantera de la cola.

La cola vacía se representa mediante la cola en la cual *rear* es igual a –1.

Este método no es eficiente. Cada supresión implica mover cada elemento restante de la cola. Si una cola contiene 1000 elementos, es evidente que este método cuesta mucho.

La operación de remover un elemento de una cola implica lógicamente la manipulación de sólo un elemento: **el que se encuentra en ese momento en la parte delantera de la cola**.

Otra solución es considerar el arreglo que contiene la cola como un círculo y no como una línea recta.

Bajo esta representación es difícil determinar cuándo está vacía la cola. La condición q.rear < q.front ya no es válida como una prueba de la cola vacía.

Una forma de solucionar este problema es establecer la convención de que el valor *q.front* es el índice de arreglo inmediatamente antes del primer elemento de la cola y no el índice del primer elemento mismo. Por tanto, dado que *q.rear* es el índice del último elemento de la cola, la *condición q.front==q.rear* implica que la cola está vacía.

Entonces, una cola puede inicializarse mediante el enunciado

```
q.front = q.rear = MAXQUEUE-1;
```

Observe que *q.front* y *q.rear* se inicializan en el último índice del arreglo y no en –1 o 0 porque, bajo esta representación, él último elemento del arreglo está inmediatamente antes que el primero dentro de la cola. Dado que *q.rear* es igual a *q.front*, la cola está vacía en un principio.

La función *empty* se recodifica como

```
int empty (struct queue *pq)
{
  return ((pq->front == pq->rear)? TRUE: FALSE);
} /* fin de empty */
```

La operación *remove (q)* se recodifica como

```
int remove (struct queue *pq)
{
  if (empty(pq)) {
     printf ("queue underflow");
     exit(1);
  } /* fin de if */
  if (pq->front == MAXQUEUE-1)
    pq->front = 0;
  else
    (pq->front)++;
  return (pq->items[pq->front]);
} /* fin de remove */
```

Observe que *pq->front*, debe actualizarse antes de extraer un elemento.

Parece que no hay un modo de distinguir entre la cola vacía y la cola llena bajo estos lineamientos.

Una solución es sacrificar un elemento del arreglo y permitir que una cola crezca hasta tener una unidad menos que el tamaño del arreglo.

La operación *insert (q, x)* se recodifica del modo siguiente:

```
void insert (struct queue *pq, int x)
{
  if (pq->rear == MAXQUEUE-1)
    pq->rear = 0;
  else
    (pq->rear)++;
  if (pq->rear == pq->front) {
    printf ("queue overflow");
    exit(1);
  } /* fin de if */
  pq->items[pq->rear] = x;
  return;
}  /* fin de insert */
```

Ocurre la prueba de desbordamiento en *insert* después de ajustar *pq->rear*. En tanto que la prueba de subdesbordamiento en *remove* ocurre inmediatamente después de introducir la rutina, antes de actualizar *pq->front*.

Listas Ligadas

Una lista es una estructura **dinámica** de datos. La cantidad de nodos en una lista puede variar de manera **drástica** conforme se **insertan** o **remueven** elementos. La naturaleza dinámica de una lista contrasta con la naturaleza **estática** de un **arreglo**, cuyo **tamaño** permanece **constante**.

En una representación secuencial, los elementos de una pila o cola están ordenados **implícitamente** por el **orden secuencial de almacenamiento**.

Ahora bien, suponga que los elementos de una pila o de una cola fueron ordenados **explícitamente**, es decir**, cada elemento contenía dentro de sí mismo la dirección del siguiente**. Tal ordenamiento explícito se conoce como una **lista ligada lineal**.

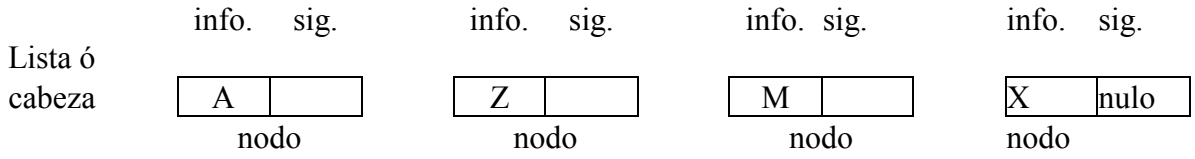
Cada elemento de la lista se denomina un **nodo** y contiene dos campos, uno de **información** y uno de **dirección siguiente**.

El campo de información contiene el elemento real en la lista. El campo de dirección siguiente contiene la dirección del siguiente nodo en la lista. Tal dirección, que se usa para accesar a un nodo particular, se conoce como **apuntador**.

Se accesa a toda la lista ligada a partir de un apuntador externo **cabeza** (*head*) que apunta al (contiene la dirección del) primer nodo en la lista.

Por **apuntador externo**, nos referimos a uno que no está incluido dentro de un nodo. Más bien, es posible accesar a su valor directamente haciendo referencia a una variable.

El campo de dirección siguiente del último nodo en la lista contiene un valor especial, conocido como **null** (**nulo**), el cual no es una dirección válida. Este **apuntador nulo** se usa para señalar el fin de una lista.



La lista que no contiene nodos se denomina **lista vacía** o **lista nula**. El valor de la lista de apuntador externo para esta lista es el apuntador nulo. Una lista se inicializa para la lista vacía mediante la operación **cabeza = null**.

Las operaciones básicas de una lista son:

- Recorrido de la lista.
- Insertar un elemento.
- Borrado de un elemento.

Recorrido de la lista

La operación de recorrido consiste en visitar cada uno de los nodos que forman la lista. La visita de un nodo puede definirse por medio de una operación muy simple (por ejemplo la impresión de la información del mismo), o por medio de operaciones tan complejas como se desee.

Para recorrer todos los nodos de una lista se comienza con el primero. Tomando el valor del campo SIG de éste se avanza al segundo; a su vez, el campo SIG del segundo nos dará acceso al tercero, y así sucesivamente. En general la dirección de un nodo, excepto el primero, está dada por el campo SIG de su predecesor.

El siguiente algoritmo presenta los pasos necesarios para recorrer iterativamente una lista. Este algoritmo recorre una lista cuyo primer nodo está apuntado por *P*. **q** es una variable de tipo puntero.

```
Recorreiterativo(P)
Inicio
        q ← p
        mientras q ≠ nulo
                Escribir info(q) // o lo que sea necesario realizar
                q ← siguiente(q)  /* apunta al siguiente nodo de la lista */
        fin_mientras
Fin
```

Como las listas son estructuras de datos recursivas, pueden manejarse fácilmente con procesos recursivos. El siguiente algoritmo es una versión recursiva del algoritmo iterativo.

Este algoritmo recorre una lista recursivamente. P es el apuntador al nodo a visitar.

```
Recorrerecursivo(P)
Inicio
        Si  p ≠ nulo
                Escribir info(p)   // o lo que sea necesario realizar
                Recorrerecursivo (siguiente(p))
        /* llamada recursiva con el apuntador al siguiente nodo de la lista */
        fin_si
Fin
```

Insertar un nodo en la parte delantera de una lista.

Obtener un nodo en el cual alojar la información. La operación

p = creaNodo();

obtiene un nodo vacío y establece el contenido de una variable nombrada *p* en la dirección de este nodo. El valor de *p* es un apuntador a un nodo recién asignado.

Insertar la información en la parte *info* del nodo recién asignado. Esto se hace mediante la operación

setInfo(p, información);

Después de establecer la parte *info* de *nodo (p)* es necesario establecer la parte *siguiente* de este nodo. Debido a que *nodo (p)* va a insertarse en la parte delantera de la lista, el nodo que sigue debe ser el primer nodo actual en la lista.

Debido a que la variable *lista* contiene la dirección de ese primer nodo, *nodo (p)* se agrega a la lista ejecutando la operación

setSiguiente (p, lista) ;

Esta operación coloca el valor de *lista* (el cual es la dirección del primer nodo en la lista) en el campo *siguiente* de *nodo (p)*.

Debido a que *lista* es el apuntador externo a la lista, su valor debe modificarse en la dirección del nuevo primer nodo de la lista. Esto se hace ejecutando la operación

lista = p;

que cambia el valor de *lista* al valor de *p*.

p se usa como una **variable auxiliar** durante el proceso de modificar la lista, pero su valor es irrelevante al estado de la lista antes y después del proceso. Una vez ejecutadas las operaciones anteriores, el valor de *p* puede cambiarse sin afectar la lista.

Resumiendo todos los pasos, tenemos **un algoritmo** para **agregar un nodo** a la **parte delantera** de la lista *lista*:

```
p = creaNodo();
setInfo (p, x) ; // (x es la información)
setSiguiente (p, lista);
lista = p;
```

Eliminar el primer nodo de una lista no vacía.

Almacenando el valor de su campo *info* en una variable *x*.

El proceso es casi lo opuesto del proceso para agregar un nodo a la parte delantera de una lista. Se ejecutan las operaciones siguientes:

```
p = lista;
lista =getSiguiente (p);
x = getInfo (p);
```

Se usa la variable *p* como **auxiliar** durante el proceso de remover el primer nodo de la lista. Pero una vez que cambia el valor de *p*, **no hay forma de accesar al nodo en absoluto**, dado que ni un apuntador externo ni un campo *siguiente* contienen su dirección.

Por tanto, **el nodo** no se usa en ese momento y **no puede volver a usarse**, aun cuando **ocupa** un valioso **espacio de almacenamiento**.

Sería preferible tener algún mecanismo para hacer que *nodo (p)* quedara disponible para reutilizarse, incluso si cambiara el valor del apuntador *p*. La operación que hace esto es

```
liberaNodo (p);
```

Una vez ejecutada esta operación, **no es válido** hacer **referencia** a *nodo (p)*, dado que el nodo ya no está asignado. Como el valor de *p* es un apuntador a un nodo que ha sido liberado, cualquier referencia a este valor **tampoco es válida**.

Sin embargo, el nodo podría reasignarse y también podría reasignarse un valor a *p* mediante la operación p = creaNodo(). Observe que decimos que el nodo "podría" reasignarse, dado que la operación creaN*odo* retorna un apuntador a un nodo recién asignado. **Nada garantiza que este nodo nuevo sea el mismo que el que se acaba de liberar**.

En realidad los nodos no se usan y reúsan sino, más bien, se **crean** y **destruyen**.

Representación ligada de pilas

La operación de agregar un elemento a la parte delantera de una lista ligada es muy similar a la de agregar un elemento a una pila. En ambos casos, se añade un nuevo elemento y es el único al que se puede accesar de inmediato en un conjunto. Una pila sólo puede accesarse mediante su elemento superior y **una lista sólo puede accesarse a partir del apuntador a su primer elemento**. Asimismo, la operación de remover el primer elemento de una lista ligada es similar a la de remover una pila.

Una pila también puede representarse mediante una lista ligada lineal. El **primer nodo** de la lista es la **parte superior** de la pila. Si un apuntador externo *s* apunta a esta lista ligada, la operación *push (s, x)* podría implementarse mediante

push (s, x)	**x = pop (s)**
p = obten_nodo (); info(p) = x; siguiente(p) = s; s = p;	**if** (empty(s)) { printf("Subdesbordamiento"); exit(1); } **else** { p = s; s = siguiente(p); x = info(p); libera_nodo(p); } /* fin de if */ **return** (x);

La operación *empty(s)* se comprueba si *s* es igual a *null*. La operación *x=pop(s)* remueve el primer nodo de la lista no vacía y señala subdesbordamiento si la lista está vacía.

Representación ligada de colas

Bajo la representación de lista ligada, una cola *q* consta de una lista y dos apuntadores, *q.front* y *q.rear*. Las operaciones *empty(q)* y *x = remove(q)*, son similares por completo a *empty(s)* y *x=pop(s)*, en donde el apuntador *q.front* sustituye a *s*. Sin embargo, se debe tener **atención especial** cuando se remueve el **último elemento** de una **cola**. En este caso, *q.rear* también debe establecerse en *null*, dado que en una cola vacía, tanto *q.front* como *q.rear* deben ser *null*. El algoritmo para *x = remove(q)* es el siguiente:

x = remove (q)	**insert (q, x)**
if (empty (q)) { printf ("Subdesbordamiento"); exit(1); } p = q.front; x = info(p); q.front = getSiguiente (p); **if** (q.front == null) q.rear = null; libera_nodo(p); **return** (x);	p = obten_nodo(); info(p) = x, setSiguiente(p, null); **if** (q.rear == null) q.front = p; **else** setSiguiente(q.rear, p); q.rear = p;

¿Cuáles son las **desventajas** de representar una **pila** o **cola** por medio de una **lista ligada**? Es evidente que **un nodo en una lista ligada ocupa más espacio de almacenamiento** que el elemento correspondiente en un arreglo, dado que en un nodo de lista se requieren dos partes de información por elemento (*info* y *siguiente*), en tanto que sólo se necesita una parte de información en la implementación de arreglo.

Otra **desventaja** es el **tiempo adicional** que se gasta en **administrar** la lista disponible.

La **ventaja** de usar listas ligadas es que todas **las pilas y colas** de un programa **tienen acceso a la misma lista de nodos libres**. Los nodos que no usa una pila pueden ser usados por otra, mientras la cantidad total de nodos en uso en cualquier momento no sea mayor que la cantidad total de nodos disponibles.

La lista ligada como estructura de datos

Las listas ligadas son importantes como estructuras de datos por derecho propio. **Un elemento se accesa en una lista ligada recorriendo la lista desde su inicio**. Una implementación de arreglo permite el acceso al enésimo elemento en un grupo usando una sola operación, en tanto que una implementación de lista requiere ***n*** operaciones. Es necesario pasar por cada uno de los primeros ***n-1*** elementos antes de llegar al **enésimo** elemento, porque **no hay una relación entre la posición de memoria que ocupa el elemento de una lista y su posición dentro de ella**.

La **ventaja** de una **lista sobre un arreglo** ocurre cuando es necesario **insertar** o **suprimir** un elemento **en medio** de un grupo de otros elementos.

Suponga que los elementos se almacenan como una lista. Si *p* apunta a un elemento de la lista, insertar un elemento nuevo después de *nodo(p)* implica **asignar un nodo**, **insertar la información** y **ajustar los apuntadores**. La cantidad de trabajo requerido es independiente del tamaño de la lista.

Suponga que *inserta_después (p, x)* denota la operación de insertar un elemento *x* en una lista después de un nodo al que se apunta mediante *p*. Esta operación se realiza del modo siguiente:

```
q = obten_nodo();
setInfo(q, x);
setSiguiente(q, getSiguiente(p) );
setSiguiente(p, q);
```

Sólo puede insertarse un elemento después de cierto nodo, no antes de él. Esto se debe a que no hay un modo de avanzar de un cierto nodo a su predecesor en una lista lineal, sin tener que recorrer la lista desde el principio.

Suponga que *borra_después (p, x)* denota la operación de suprimir el nodo después de *nodo(p)* y asignar su contenido a la variable *x*. Esta operación se efectúa del modo siguiente:

```
q = getSiguiente(p);
x = getInfo(q);
setSiguiente(p, getSiguiente(q) );
libera_nodo(q);
```

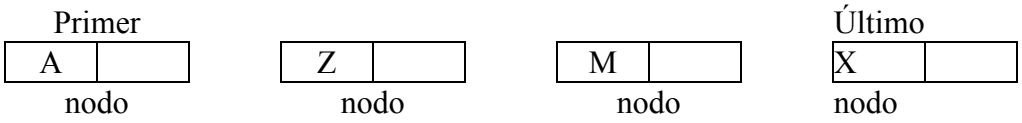
A continuación se presenta una tabla que resume las principales **funciones u operaciones** que se le pueden aplicar a una **lista ligada lineal**:

	Inserta nodo	**Borra nodo**
Lista ligada lineal:		
Al **comienzo** de la lista	p = obten_nodo (); setInfo(p, x); setSiguiente(p, s); s = p;	**if** (empty(s)) { printf("Subdesbordamiento"); exit(1); } **else** { p = s; s = getSiguiente(p); x = getInfo(p); libera_nodo(p); } /* fin de if */ **return** (x);
Al **final** de la lista	p = obten_nodo(); setInfo(p, x); setSiguiente(p, null); **if** (q.rear == null) q.front = p; **else** setSiguiente(q.rear, p); q.rear = p;	**//revisar** **if** (empty (q)) { printf ("Subdesbordamiento"); exit(1); } p=q.rear; x=getInfo(p); if(q.front==q.rear) q.front=q.rear=null; else{ temp=q.front; while(getSiguiente(temp)!=p) temp=getSiguiente(temp); setSiguiente(temp, null); q.rear=temp; } libera_nodo(p); return x;
En **medio** de la lista	q = obten_nodo(); setInfo(q, x); setSiguiente(q, getSiguiente(p)); setSiguiente(p, q);	q = getSiguiente(p); x = getInfo(q); setSiguiente(p, getSiguiente(q)); libera_nodo(q);

Listas Circulares

Una **lista circular** es aquélla en la que el campo *siguiente* en el **último** nodo contiene un **apuntador de regreso** al **primer** nodo y **no** a un **apuntador nulo**. Desde cualquier punto en esta lista es posible llegar a cualquier otro punto de ella. Si empezamos en cierto nodo y recorremos toda la lista, terminamos en el punto inicial.

Una lista circular **no tiene** un primer o último nodo propiamente. Por tanto, debemos establecer un primer y último nodo por **convención**. Una convención útil es permitir que el **apuntador externo** a la lista circular apunte al **último** nodo y permitir que el **siguiente** nodo sea el **primero**. Ver la figura:



Primero y último nodos de una lista circular

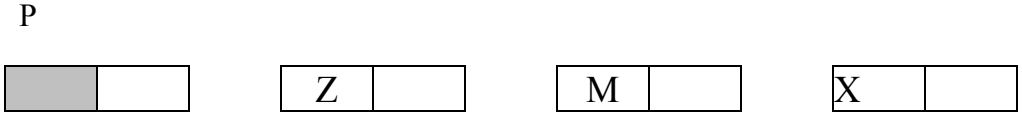
Si *p* es un apuntador externo para una lista circular, esta convención establecida permite el acceso al último nodo de la lista haciendo referencia a *nodo(p),* y al primer nodo de la lista, haciendo referencia a *nodo(getSiguiente(p))*.

Esta convención tiene la ventaja de poder agregar o remover cualquier elemento en forma conveniente desde la parte delantera o posterior de una lista.

También se establece la convención de que un **apuntador nulo** representa una **lista circular vacía**.

Las operaciones en **listas circulares** son **similares** a las operaciones **en listas lineales**.

En el caso de la operación de recorrido de listas circulares, es necesario aclarar que se debe considerar algún **criterio** para **detectar** cuándo se han **visitado todos los nodos** para **evitar** caer en **ciclos infinitos**. Una posible solución consiste en usar un **nodo extra**, llamado **nodo de cabecera**, para indicar el **inicio de la lista**. Este nodo contendrá **información especial**, de tal manera que se **distinga** de los demás y así podrá hacer **referencia al principio** de la lista. La siguiente figura presenta una lista circular con nodo de cabecera:



Lista circular con nodo de cabecera

Una solución más simple resultaria de comparar el final de la lista con al apuntador externo, que apunta precisamente al último elemento.

Listas doblemente ligadas circulares

La principal ventaja de las listas circulares es que permiten la **navegación** en **cualquier sentido** a través de la misma y además, se puede **recorrer** toda **la lista** partiendo de **cualquier nodo**. Sin

embargo, debemos hacer notar que se deben establecer condiciones adecuadas para detener el recorrido de una lista para **evitar caer en ciclos infinitos**. Lo mismo que en el caso de listas circulares podría usarse un nodo de cabecera. Este nodo tendría las características descritas anteriormente, y serviría como referencia para detectar cuándo se ha recorrido totalmente la lista.

Cabecera

Lista doblemente ligada circular con nodo de cabecera

La estructura del nodo sería la siguiente:

```
struct nodo {
     char info;
     struct nodo *aptSig;
     struct nodo *aptAnt;
};
```

Operaciones:
- Recorrido de la lista hacia adelante, hacia atrás.
- Inserción al inicio, final, antes o después de un nodo.
- Borrado al inicio, al final, en alguna posición, antes o después de un nodo.

Resumen de listas ligadas.

Se han considerado las siguientes listas ligadas:

1. Simplemente ligadas	1.1. Lineales
	1.2. Circulares
2. Doblemente ligadas	2.1. Lineales
	2.2. Circulares.

```
/* Ejemplo del manejo de una lista ligada */
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>

struct nodo {                    /* estructura o nodo auto-referenciado */
      char info;
      struct nodo *aptsig;
};

typedef struct nodo ListaNodos;
typedef ListaNodos *aptListaNodos;

void Inserta_nodo (aptListaNodos *, char);
char Borra_nodo (aptListaNodos *, char);
int Lista_vacia (aptListaNodos);
void Despliega_lista (aptListaNodos);
void Despliega_menu (void);

main ()
{
      aptListaNodos aptinicio = NULL;
      int opcion;
      char elemento;

      Despliega_menu();          /* despliega el menú de opciones del programa*/
      printf("? ");
      scanf("%d", &opcion);

      while (opcion != 3) {

            switch (opcion) {
                  case 1:
                        printf("Introduzca un caracter: ");
                        scanf("\n%c", &elemento);
                        Inserta_nodo(&aptinicio, elemento);
                        Despliega_lista(aptinicio);
                        break;
                  case 2:
                        if (!Lista_vacia(aptinicio)) {
                              printf("Introduzca el caracter a borrar: ");
                              scanf("\n%c", &elemento);

                              if (Borra_nodo(&aptinicio, elemento)) {
                                    printf("%c borrado.\n", elemento);
                                    Despliega_lista(aptinicio);
                              }
                              else
                                    printf("%c no encontrado.\n\n", elemento);
                        }
                        else
                              printf("La lista esta vacía.\n\n");
                        break;
                  default:
                        printf("Selección inválida.\n\n");
                        Despliega_menu();
                        break;
                  }
                  printf("? ");
                  scanf("%d", &opcion);
            }
            printf("Fin de programa.\n");
```

```
            return 0;
}

/* Imprime las instrucciones del menú de opciones */
void Despliega_menu(void)
{
      printf("Introduzca su opcion: \n"
                  "    1 para insertar un nodo a un elemento en la lista.\n"
                  "    2 para borrar un elemento de la lista.\n"
                  "    3 para terminar.\n");
}

/* Inserta un nuevo nodo en la lista de manera ordenada */
void Inserta_nodo(aptListaNodos *sPtr, char dato)
{
      aptListaNodos aptnuevo, aptprevio, aptactual;

      aptnuevo = malloc(sizeof(ListaNodos));

      if (aptnuevo != NULL) {           /* hay espacio disponible de RAM */
            aptnuevo->info = dato;
            aptnuevo->aptsig = NULL;

            aptprevio = NULL;
            aptactual = *sPtr;

            while (aptactual != NULL && dato > aptactual->info) {
                  aptprevio = aptactual;      /* avanza hacia el nodo próximo */
                  aptactual = aptactual->aptsig;
            }

            if (aptprevio == NULL) {
                  aptnuevo->aptsig = *sPtr;
                  *sPtr = aptnuevo;
            }
            else {
                  aptprevio->aptsig = aptnuevo;
                  aptnuevo->aptsig = aptactual;
            }
      }
      else
            printf("%c no insertado. No memoria disponible.\n", dato);
}

/* Borrar un elemento de una lista */
char Borra_nodo(aptListaNodos *sPtr, char dato)
{
      aptListaNodos aptprevio, aptactual, apttemp;

      if (dato == (*sPtr)->info) {
            apttemp = *sPtr;
            *sPtr = (*sPtr)->aptsig;             /* liga el nodo */
            free(apttemp);                          /* libera el nodo */
            return dato;
      }
      else {
            aptprevio = *sPtr;
            aptactual = (*sPtr)->aptsig;

            while (aptactual != NULL && aptactual->info != dato) {
                  aptprevio = aptactual;    /* avanza hacia el siguiente nodo */
                  aptactual = aptactual->aptsig;
            }
```

```
            if (aptactual != NULL) {
                  apttemp = aptactual;
                  aptprevio->aptsig = aptactual->aptsig;
                  free(apttemp);
                  return dato;
            }
      }
      return '\0';
}


/* Regresa 1 (TRUE) si la lista esta vac¡a, 0 (FALSE) en otro caso */
int Lista_vacia(aptListaNodos sPtr)
{
      return sPtr == NULL;
}

/* Imprime la lista comenzando por el principio de la misma*/
void Despliega_lista(aptListaNodos aptactual)
{
      if (aptactual == NULL)
            printf("La lista esta vacia.\n\n");
      else {
            printf("La lista es:\n");

            while (aptactual != NULL) {
                  printf("%c  --> ", aptactual->info);
                  aptactual = aptactual->aptsig;
            }
            printf("NULL\n\n");
      }
}
```

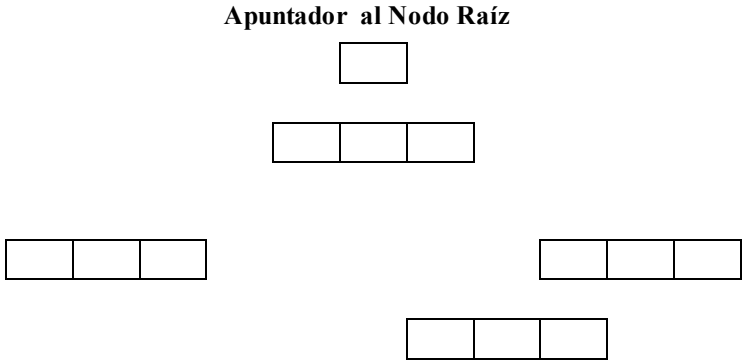
Árboles Binarios

Un **árbol** es una estructura **no lineal** y de **dos dimensiones** de datos, con **propiedades especiales**. Los nodos de los árboles contienen **dos** o **más** enlaces.

Los **árboles binarios**, son árboles cuyos nodos contienen como máximo **dos enlaces**, es decir, ninguno, uno, o ambos de los cuales pudieran ser nulos.

El **nodo raíz** es el **primer nodo** de un árbol. Cada enlace en el nodo raíz se refiere a un **hijo**. El **hijo izquierdo** es el **primer nodo** en el **subárbol izquierdo** y el **hijo derecho** es el **primer nodo** en el **subárbol derecho**.

Los **hijos de un nodo** se conocen como **descendientes**. Un **nodo sin hijos** se conoce como **nodo de hoja.**



Representación gráfica de un árbol binario

Otra definición de árbol binario:

"Un árbol binario es un conjunto finito de elementos que está vacío o dividido en tres subconjuntos separados. El primer subconjunto contiene un elemento único llamado la **raíz** del árbol. Los otros dos subconjuntos son por sí mismos árboles binarios y se les conoce como **subárboles izquierdo** y **derecho** del árbol original. Un subárbol izquierdo o derecho puede estar vacío. Cada elemento de un árbol binario se denomina nodo del árbol."

Si A es la raíz de un árbol binario y B es la raíz de su subárbol izquierdo o derecho, se dice que A es el **padre** de B y se dice que B es el **hijo izquierdo** o **derecho** de A.

El nodo n1 es un **ancestro** del nodo n2, si n1 es el padre de n2 o el padre de algún ancestro de n2.

Un nodo n2 es un **descendiente izquierdo** del nodo n1, si n2 es el hijo izquierdo de n1 o un descendiente del hijo izquierdo de n1. Un **descendiente derecho** se define de la misma forma.

Dos nodos son **hermanos** si son los hijos izquierdo y derecho del mismo padre.

Aunque los árboles naturales crecen con sus raíces en el suelo y sus hojas en el aire, los **computólogos** casi universalmente representan las estructuras de árbol con la **raíz** en la **parte superior** y las **hojas** en la **parte inferior**.

La dirección de la raíz a las hojas es **"hacia abajo"** y la dirección opuesta es **"hacia arriba".** Pasar de las hojas a la raíz se denomina **"subir"** por el árbol, y dirigirse de la raíz a las hojas se denomina **"descender"** por el árbol.

Si cada nodo que no es hoja en un árbol binario tiene subárboles izquierdo y derecho que no están vacíos, el elemento se clasifica **como árbol estrictamente binario**. Un árbol estrictamente binario con *n* hojas siempre contiene *2n-1* nodos.

El **nivel** de un nodo en un árbol binario se define del modo siguiente: la raíz del árbol tiene el nivel 0, y el nivel de cualquier otro nodo en el árbol es uno más que el nivel de su padre.

La **profundidad** de un árbol binario es el máximo nivel de cualquier hoja en el árbol. Esto es igual a la longitud de la trayectoria más larga de la raíz a cualquier hoja. Un **árbol binario completo** de profundidad *d* es un árbol estrictamente binario que tiene todas sus hojas en el nivel *d*.

```
/* Crea un árbol binario y lo recorre en preorden, inorden y postorden */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <alloc.h>
#include <conio.h>

struct NodoArbol {
        struct NodoArbol *aptIzq;
        int dato;
        struct NodoArbol *aptDer;  };

typedef struct NodoArbol NODOARBOL;
typedef NODOARBOL *APTNODOARBOL;
```

```
void insertaNodo (APTNODOARBOL *, int);
void inOrden (APTNODOARBOL);
void preOrden (APTNODOARBOL);
void postOrden (APTNODOARBOL);

main()
{
        int i, item;
        APTNODOARBOL aptRaiz = NULL;

        srand (time(NULL));
        /* Intenta insertar 10 valores aleatorios entre 0 y 14 en el árbol */  printf("Los números
colocados en el árbol son:\n");
        for (i=1; i<=10; i++) {                 item = rand() % 15;             printf("%3d", item);
        insertaNodo (&aptRaiz, item);   }

        /* Recorre el árbol en orden preOrden */
        printf("\n\nEl recorrido preOrden es:\n");
        preOrden (aptRaiz);

        /* Recorre el árbol en orden inOrden */
        printf("\n\nEl recorrido inOrden es:\n");
        inOrden (aptRaiz);

        /* Recorre el árbol en orden postOrden */
        printf("\n\nEl recorrido postOrden es:\n");
        postOrden (aptRaiz);

        return 0;
}
```

```
void insertaNodo (APTNODOARBOL *aptArbol, int valor) {
        if (*aptArbol == NULL) {   /* aptArbol es NULO */
                *aptArbol = malloc(sizeof(NODOARBOL));

                if (*aptArbol != NULL) {
                        (*aptArbol)->dato = valor;
                        (*aptArbol)->aptIzq = NULL;
                        (*aptArbol)->aptDer = NULL;
                }
                else
                        printf ("%d no insertado. No memoria disponible.\n", valor);
        }
        else
                if (valor < (*aptArbol)->dato)
                        insertaNodo (&((*aptArbol)->aptIzq), valor);
                else
                        if (valor > (*aptArbol)->dato)
                                insertaNodo (&((*aptArbol)->aptDer), valor);
                        else
                                printf (" (valor duplicado = %3d) ", valor);
}

void inOrden (APTNODOARBOL aptArbol) {
        if (aptArbol != NULL) {
                inOrden (aptArbol->aptIzq);
                printf ("%3d", aptArbol->dato);
                inOrden (aptArbol->aptDer);
                }
}

void preOrden (APTNODOARBOL aptArbol)
{
        if (aptArbol != NULL) {
                printf ("%3d", aptArbol->dato);
                preOrden (aptArbol->aptIzq);
                preOrden (aptArbol->aptDer);
                }
}

void postOrden (APTNODOARBOL aptArbol) {
        if (aptArbol != NULL) {
                postOrden (aptArbol->aptIzq);
                postOrden (aptArbol->aptDer);
                printf ("%3d", aptArbol->dato);
                }
}
```

Árbol binario de búsqueda

Los árboles binarios de búsqueda (que **no** tienen **valores duplicados** de nodos) tienen la característica que los valores en cualquier **subárbol izquierdo** son **menores** que el valor en sus nodos **padre**, y los valores en **cualquier subárbol derecho** son **mayores** que el valor en sus nodos **padre**.

Note que la forma del **árbol binario de búsqueda** que corresponde a un conjunto de datos **puede variar**, dependiendo del **orden** en el cual **los valores están insertados** dentro del árbol.

Las funciones utilizadas para crear un árbol binario de búsqueda y recorrer el árbol, son **recursivas**.

En un árbol binario de búsqueda un nodo puede ser únicamente insertado como nodo de hoja.

Las funciones **inOrden**, **preOrden** y **postOrden** cada una de ellas recibe un árbol (es decir, el apuntador al nodo raíz del árbol, por valor) y recorren el árbol.

Un recorrido inOrden es:

1. Recorrer el subárbol izquierdo inOrden.
2. Procesar el valor en el nodo.
3. Recorrer el subárbol derecho inOrden.

El valor en un nodo no es procesado en tanto no sean procesados los valores de su subárbol izquierdo. Note que el recorrido **inOrden** de un árbol de búsqueda binario **imprime los valores** de los nodos en forma **ascendente**. El proceso de crear un árbol de búsqueda binario, de hecho ordena los datos, y por lo tanto este proceso se llama la **clasificación de árbol binario**.

Un recorrido preOrden es:

1. Procesar el valor en el nodo.
2. Recorrer el subárbol izquierdo preOrden.
3. Recorrer el subárbol derecho preOrden.

El valor en cada nodo es procesado conforme se pasa por ese nodo. Después de que se procese el valor en un nodo dado, son procesados los valores del subárbol izquierdo, y a continuación los valores en el subárbol derecho.

Un recorrido postOrden es:

1. Recorrer el subárbol izquierdo postOrden.
2. Recorrer el subárbol derecho postOrden.
3. Procesar el valor en el nodo.

El valor en cada nodo no se imprime hasta que sean impresos los valores de sus hijos.

El árbol binario de búsqueda facilita la eliminación de duplicados.

Conforme se crea el árbol, cualquier intento para insertar un valor duplicado será detectado porque en cada una de las comparaciones un duplicado seguirá las mismas decisiones "ir a la izquierda" o "ir a la derecha" que utilizó el valor original. Entonces, el duplicado eventualmente será comparado con un nodo que contenga el mismo valor. Llegado a este punto **el valor duplicado** pudiera simplemente ser **descartado**.

También es **rápido buscar en un árbol binario un valor** que **coincida** con un valor **clave**.

Si el árbol es denso, entonces cada nivel contendrá aproximadamente dos veces tantos elementos como el nivel anterior. Por lo tanto un árbol binario de búsqueda con *n* elementos tendría un máximo de $\log_2 n$ (logaritmo de base 2 de *n* niveles) y, por lo tanto, tendrían que efectuarse un máximo **$\log_2 n$ de comparaciones**, ya sea para **encontrar una coincidencia**, o para **determinar que no existe** ninguna.

El **árbol binario de búsqueda** es una estructura sobre la cual se pueden realizar **eficientemente** las operaciones de **búsqueda**, **inserción** y **eliminación**.

Comparando esta estructura con otras, pueden observarse ciertas ventajas:

- En un **arreglo** es posible **localizar datos** eficientemente si los mismos se encuentran **ordenados**, pero las operaciones de **inserción** y **eliminación** resultan **costosas**.

- En las **listas**, las operaciones de **inserción** y **eliminación** se pueden llevar a cabo con **facilidad**, sin embargo la **búsqueda** es una operación bastante **costosa** que incluso nos puede llevar a recorrer todos los elementos de ella para localizar uno en particular.

Formalmente se define un **árbol binario de búsqueda** de la siguiente manera:

"Para todo nodo *T* del árbol debe cumplirse que todos los valores de los nodos del subárbol izquierdo de *T* deben ser menores al valor del nodo *T*. De forma similar, todos los valores de los nodos del subárbol derecho de *T* deben ser mayores al valor de nodo *T*".

Algoritmos de búsqueda.

El siguiente algoritmo localiza un nodo en un árbol binario de búsqueda.

Raíz es una variable de tipo puntero que apunta a la raíz del árbol.

Dato es una variable de tipo entero que contiene la información que se desea localizar en el árbol. Cabe aclarar que en la primera llamada la variable *Raíz* no puede ser nula.

```
Búsqueda (Raíz, Dato)
Inicio
        Si (Dato < info(Raíz))
                Si (izquierdo(Raíz) = nulo)
                        Escribir "El nodo no se encuentra en el árbol"
                 si no
                        Búsqueda (izquierdo(Raíz), Dato))
                fin_si
         si no
                Si (Dato > info(Raíz))
                                Si (derecho(Raíz) = nulo)
                                Escribir " El nodo no se encuentra en el árbol"
                          si no
                                Búsqueda (derecho(Raíz, Dato))
                        fin_si
                 si no
                        Escribir "El nodo se encuentra en el árbol"
                fin_si
        fin_si
Fin
```

Otra variante del algoritmo de búsqueda es el siguiente:

```
Búsqueda1 (Raíz, Dato)
Inicio
        Si Raíz ≠ nulo
                Si (Dato < info(Raíz))
                                Búsqueda1 (izquierdo(Raíz), Dato))
                        si no
                                Si (Dato > info(Raíz))
                                Búsqueda1 (derecho(Raíz, Dato))
                                si no
                                Escribir "El nodo se encuentra en el árbol"
                        fin_si
                        fin_si
        si no
                        Escribir "El nodo no se encuentra en el árbol"
        fin_si
Fin
```

Inserción en un árbol binario de búsqueda

La **inserción** es una operación que se puede realizar **eficientement**e en un **árbol binario de búsqueda**. La estructura crece conforme se inserten elementos al árbol. Los pasos que deben realizarse para insertar un elemento a un árbol binario de búsqueda son los siguientes:

1. Debe compararse la clave a insertar con la raíz del árbol. Si es mayor, debe avanzarse hacia el subárbol derecho. Si es menor, debe avanzarse hacia el subárbol izquierdo.
2.
3. Repetir sucesivamente el paso 1 hasta que se cumpla alguna de las siguientes condiciones:
   2.1. El subárbol derecho es igual o vacío, o el subárbol izquierdo es igual a vacío; en cuyo caso se procederá a insertar el elemento en el lugar que le corresponde.
   2.2. La clave que quiere insertarse es igual a la raíz del árbol; en cuyo caso no se realiza la inserción.

El siguiente algoritmo realiza la inserción de un nodo en un árbol binario de búsqueda. *Raíz* es una variable de tipo puntero y la primera vez debe ser distinta de vacío. *Dato* es una variable de tipo entero que contiene la información del elemento que se quiere insertar. Se utiliza además, como auxiliar, la variable o*tro* de tipo puntero.

```
Inserción (raíz, dato)
Inicio
        Si dato < getInfo(raíz)
                Si getIzquierdo(raíz) = nulo
                        generaNodo(otro)
                                setIzquierdo(otro, nulo)
                                setDerecho(otro, nulo)
                                setInfo(otro, dato)
                                setIzquierdo(raíz, otro)
                si no
                        Inserción (getIzquierdo(raíz), dato)
        si no
                        Si dato >getIinfo(raíz)
                                Si getDerecho(raíz) = nulo
                                generaNodo(otro)
                                        setIzquierdo(otro, nulo)
                                setDerecho(otro, nulo)
                                setInfo(otro, dato)
                                setDerecho(raíz, otro)
                        si no
                                Inserción (getDerecho(raíz), dato)
                                fin_si
                        si no
                                Escribir "El nodo ya se encuentra en el árbol"
                        fin_si
        fin_si
Fin
```

Otra forma de expresar el algoritmo de inserción es la siguiente:

```
Inserción1(raíz, dato)
Inicio
        Si Raíz ≠ nulo
                Si dato <getInfo(raíz)
                        Inserción1 (getIzquierdo(raíz), dato)
                si no
                        Si Dato > getInfo(raíz)
                                Inserción1 (getDerecho(raíz), dato)
                        si no
                                Escribir "El nodo ya se encuentra en el árbol"
                        fin_si
                fin_si
        si no
                        generaNodo(otro)
                        setIzquierdo(otro, nulo)
                        setDerecho(otro, nulo)
                        setIinfo(otro, dato)
                        raíz ← otro
        fin_si
Fin
```

Borrado en un árbol binario de búsqueda

La operación de **borrado** es un poco **más complicada que la de inserción**.

Ésta consiste en **eliminar un nodo del árbol sin violar los principios que definen justamente un árbol binario de búsqueda**. Se debe distinguir los siguientes casos:

1. Si el elemento a borrar es terminal u hoja, simplemente se suprime.

2. Si el elemento a borrar tiene un solo descendiente, entonces tiene que sustituirse por ese descendiente.

3. Si el elemento a borrar tiene los dos descendientes, entonces se tiene que sustituir por el nodo que se encuentra más a la izquierda en el subárbol derecho o por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

Además, debemos **recordar que antes de eliminar un nodo, debe localizársele en el árbol**. Para esto, se utilizará el algoritmo de búsqueda presentado anteriormente.

El siguiente algoritmo realiza la eliminación de un elemento en un árbol binario de búsqueda.

Raíz es una variable de tipo puntero (por referencia).
Dato es un variable de tipo entero que contiene la información del nodo que se desea eliminar. Además *Aux*, *Aux1* y *Otro* son variables auxiliares de tipo puntero.

```
Eliminación (Raíz, Dato)
Inicio
        Si Raíz ≠ nulo
                Si Dato < getInfo(Raíz)
                        Eliminación (getIzquierdo(Raíz), Dato)
                si no
                        Si Dato > getInfo(Raíz)
                                Eliminación (getDerecho(Raíz), Dato)
                        si no
                                Otro ← Raíz
                                Si getDerecho(Otro) = nulo
                                        Raíz ← getIzquierdo(Otro)
                                si no
                                        Si getIzquierdo(Otro) = nulo
                                                Raíz ← getDerecho(Otro)
                                        si no
                                                Aux ← getIzquierdo(Otro)
                                                Aux1 ← Aux
                                                Mientras getDerecho(Aux) ≠ nulo
                                                        Aux1 ← Aux
                                                        Aux ← getDerecho(Aux)
                                                fin_mientras
                                                setInfo(Otro, getInfo(Aux) )
                                                Otro ← Aux
                                                setDerecho(Aux1, getIzquierdo(Aux))
                                        fin_si
                                fin_si
                        fin_si
                fin_si
                liberaNodo(Otro)
        si no
                        Escribir "El nodo no se encuentra en el árbol"
        fin_si
Fin
```

Árboles balanceados

Cuando un árbol binario crece o decrece **descontroladamente**, el rendimiento puede disminuir considerablemente. El caso más **desfavorable** se produce cuando se inserta un conjunto de **claves ordenadas** en forma **ascendente** o **descendente**.

El número promedio de comparaciones que debe realizarse para localizar una determinada clave, en un árbol binario de búsqueda con crecimiento descontrolado es *N/2*, cifra que muestra un rendimiento muy pobre en la estructura.

Con el objeto de **mejorar** el rendimiento en la búsqueda surgen los **árboles balanceados**. La idea central de éstos es la de realizar reacomodos o balanceos, después de inserciones o eliminaciones de elementos. Estos árboles también reciben el nombre de **AVL** en honor a sus inventores, dos matemáticos rusos, G.M. **A**delson-**V**elskii y E.M. **L**andis.

Formalmente se define un **árbol balanceado** como un **árbol binario de búsqueda**, en el cual se debe cumplir la siguiente condición: "Para todo nodo T del árbol, la altura de los subárboles izquierdo y derecho **no** debe **diferir** en **más** de una **unidad**".

Los **árboles balanceados** se parecen mucho, en su mecanismo de formación, a los **números Fibonacci**. El árbol de **altura 0 es vacío**, el árbols de **altura 1** tiene **un único nodo** y en general el número de nodos del árbol con **altura h>1** se calcula aplicando la siguiente fórmula recursiva:

$$K_h = K_{h-1} + 1 + K_{h-2}$$

donde *K* indica el número de nodos del árbol y *h* la altura.

Algunos estudios demuestran que la altura de un árbol balanceado de *n* nodos nunca excederá de 1.44*log *n*.

Al insertar un elemento en un árbol balanceado deben distinguirse los siguientes casos:

1. Las ramas izquierdas (RI) y derecha (RD) del árbol tienen la misma altura ($H_{RI} = H_{RD}$), por lo tanto:
   2|.1. Si se inserta un elemento en RI entonces H_{RI} será mayor a H_{RD}.
   2|.2. Si se inserta un elemento en RD entonces H_{RD} será mayor a H_{RI}.

Obsérvese que en cualquiera de los dos casos mencionados (1.1 y 1.2), no se viola el criterio de equilibrio del árbol.

1. Las ramas izquierda (RI) y derecha (RD) del árbol tienen altura diferente ($H_{RI} \neq H_{RD}$):
   2|.1. Supóngase que $H_{RI} < H_{RD}$:
      2.1.1. Si se inserta un elemento en RI entonces H_{RI} será igual a H_{RD}.
      2.1.2. {Las ramas tienen ala misma altura, por lo que se mejora el equilibrio del árbol}
      2.1.3. Si se inserta un elemento en RD entonces se rompe el criterio de equilibrio del árbol y es necesario restructurarlo.
   2|.1. Supóngase que $H_{RI} > H_{RD}$:
      2.1.1. Si se inserta un elemento en RI, entonces se rompe el criterio de equilibrio del árbol y es necesario restructurarlo.
      2.1.2. Si se inserta un elemento en RD, entonces H_{RD} será igual a H_{RI}.
      2.1.3. {Las ramas tienen la misma altura, por lo que se mejora el equilibrio del árbol}

Para poder determinar si un árbol está balanceado o no, debe manejarse información relativa al equilibrio de cada nodo del árbol. Surge así el concepto de **factor de equilibrio de un nodo (FE)** que se define como: la altura del subárbol derecho menos la altura del subárbol izquierdo.

$$\mathbf{FE = H_{RD} - H_{RI}}$$

Los valores que puede tomar **FE** son: **-1, 0, 1**. Si FE llegara a tomar los valores de **–2** o **2**, entonces debería **restructurarse** el árbol.

A continuación se presenta la definición de un árbol balanceado en lenguaje C.

```
struct NodoArbol {
        int dato;
        int FE;
        struct NodoArbol *aptIzq, *aptDer;
};
```

RESTRUCTURACIÓN DEL ÁRBOL BALANCEADO

El proceso de inserción en un árbol balanceado es sencillo pero con algunos detalles un poco complicados. Primero debe seguirse el camino de búsqueda del árbol, hasta localizar el lugar donde hay que insertar el elemento. Luego se calcula su FE, que obviamente será 0, y regresamos por el camino de búsqueda calculando el FE de los

distintos nodos. Si en alguno de los nodos se viola el criterio de equilibrio entonces debe restructurarse el árbol. El proceso termina al llegar a la raíz del árbol, o cuando se realiza la restructuración del mismo; en cuyo caso no es necesario determinar el FE de los restantes nodos.
Restructurar el árbol significa rotar los nodos del mismo. La rotación puede ser simple o compuesta. El primer caso involucra dos nodos y el segundo caso afecta a 3. Si la rotación es simple puede realizarse por las ramas derechas (DD) o por las ramas izquierdas (II). Si la rotación es compuesta puede realizarse por las ramas derecha e izquierda (DI) o por las ramas izquierda y derecha (ID).

ROTACIÓN
- SIMPLE
  - DD
  - II
- COMPUESTA
  - DI
  - ID

El algoritmo inserta un elemento en un árbol balanceado. Raíz es una variable de tipo puntero (por referencia). Bo es una variable de tipo booleano (por referencia). Bo se utiliza para indicar que la altura del árbol ha crecido, su valor inicial es FALSO. Dato es una variable de tipo entero que contiene la información del elemento que queremos insertar. Otro, Nodo1 y Nodo2 son variables auxiliares de tipo puntero.

```
InsertaBalanceado (Raíz, Bo, Dato)
Inicio
Si Raíz ≠ nulo
        entonces
        Si Dato < info(Raíz)
                entonces
                InsertaBalanceado (izquierdo(Raíz), Bo, Dato)
                Si Bo = VERDADERO
                        entonces
                        Según FE(Raíz)
                                = 1 :   FE(Raíz) ← 0
                                        Bo ← FALSO
                                = 0 :   FE(Raíz) ← -1
                                =-1 :   {Reestructuración del árbol}
                                        Nodo1 ← izquierdo(Raíz)
                                        Si FE(Nodo1) ≤ 0
                                                entonces { Rotación II }
                                                izquierdo(Raíz) ← derecho(Nodo1)
                                                derecho(Nodo1) ← Raíz
                                                FE(Raíz) ← 0
```

```
                                Raíz ← Nodo1 { Termina la rotación II }
                            si no {Rotación ID}
                                Nodo2 ← derecho(Nodo1)
                                izquierdo(Raíz) ← derecho(Nodo2)
                                derecho(Nodo2) ← Raíz
                                derecho(Nodo1) ← izquierdo(Nodo2)
                                izquierdo(Nodo2) ← Nodo1
                                Si FE(Nodo2) = -1
                                    entonces
                                    FE(Raíz) ← 1
                                si no
                                    FE(Raíz) ← 0
                                fin_si
                                Si FE(Nodo2) = 1
                                    entonces
                                    FE(Nodo1) ← -1
                                si no
                                    FE(Nodo1) ← 0
                                fin_si
                                Raíz ← Nodo2    { Termina la rotación ID }
                            fin_si
                            FE(Raíz) ← 0
                            Bo ← FALSO
        fin_según
si no
        Si Dato > info(Raíz)
            entonces
            InsertaBalanceado (derecho(Raíz), Bo, Dato)
            Si Bo = VERDADERO
                entonces
                Según FE(Raíz)
                    = -1 :  FE(Raíz) ← 0
                            Bo ← FALSO
                    = 0 :   FE(Raíz) ← 1
                    =1 :    {Reestructuración del árbol}
                            Nodo1 ← derecho(Raíz)
                            Si FE(Nodo1) ≥ 0
                                entonces { Rotación DD }
                                derecho(Raíz) ← izquierdo(Nodo1)
                                izquierdo(Nodo1) ← Raíz
                                FE(Raíz) ← 0
                                Raíz ← Nodo1  {Termina la rotación DD}
                            si no {Rotación DI}
                                Nodo2 ← izquierdo(Nodo1)
                                derecho(Raíz) ← izquierdo(Nodo2)
                                izquierdo(Nodo2) ← Raíz
```

```
                                                izquierdo(Nodo1) ← derecho (Nodo2)
                                                derecho (Nodo2) ← Nodo1
                                                Si FE(Nodo2) = 1
                                                        entonces
                                                        FE(Raíz) ← -1
                                                si no
                                                        FE(Raíz) ← 0
                                                fin_si
                                                Si FE(Nodo2) = -1
                                                        entonces
                                                        FE(Nodo1) ← 1
                                                si no
                                                        FE(Nodo1) ← 0
                                                fin_si
                                                Raíz ← Nodo2    {Termina la rotación DI}
                                        fin_si
                                        FE(Raíz) ← 0
                                        Bo ← FALSO
                                fin_según
                        si no
                Escribir "El nodo ya se encuentra en el árbol"
                        fin_si
                fin_si
        fin_si
    si no
        obten_nodo(Raíz) {Crear un nuevo nodo}
        info(Raíz)  ← Dato
        izquierdo(Raíz) ← nulo
        derecho(Raíz) ← nulo
        FE(Raíz) ← 0
        Bo ← VERDADERO
    fin_si
fin_si
Fin
```

Métodos de Ordenación Interna.

Es la clasificación u ordenación de datos que se encuentran en la memoria principal de la computadora.

Se asume que los datos están almacenados en una estructura de datos de libre acceso, como puede ser un vector o un arreglo.

Ordenación por burbuja.

También conocido como de intercambio directo. Se basa en comparar elementos adyacentes de la lista de datos e intercambiar sus valores si están desordenados. Se dice que los valores más pequeños burbujean hacia la parte superior de la lista, mientras que los valores más grandes se hunden hacia el fondo de la lista.

Los pasos generales del algoritmo son:

1. Se **compara** cada elemento de la lista con su valor **adyacente**, si están desordenados se **intercambian** entre sí. Se comienza comparando el primer elemento con el segundo; luego el segundo con el tercero; y se prosigue hasta el final de la lista de datos.

Al terminar de hacer un recorrido de la lista, el elemento más grande se encuentra al fondo de la lista, y algunos elementos pequeños se han acercado al principio de la misma.

2. Se **vuelve a recorrer la lista**, intercambiando los valores desordenados, pero omitiendo comparar el elemento mayor pues ya está ordenado.
3. Se siguen haciendo recorridos asta que la lista esté ordenada. El **número de recorridos** será de n-1.

Se puede optimizar para que suspenda los ordenamientos si la lista se encuentra ordenada antes del recorrido n-1.

Método de la Sacudida (shaker sort)

Este método es una optimización del método de intercambio directo o burbuja. La idea básica de este algoritmo consiste en mezclar las dos formas en que se puede realizar el método de la burbuja.

En este algoritmo cada pasada tiene dos etapas. En la primera etapa **"de derecha a izquierda"** se trasladan los elementos **más pequeños** hacia la parte **izquierda del arreglo**, almacenando en una variable la posición del último elemento intercambiado.

En la segunda etapa "**de izquierda a derecha"** se trasladan los elementos **más grandes** hacia la parte **derecha del arreglo**, almacenando en otra variable la posición del último elemento intercambiado.

Las sucesivas pasadas trabajan con los componentes del arreglo comprendidos entre las posiciones almacenadas en las variables. El algoritmo termina cuando en una etapa **no se producen intercambios** o bien, cuando **el valor del índice que almacena el extremo izquierdo del arreglo es mayor que el valor del índice que almacena el extremo derecho**.

Ordenación por Inserción Directa

El método de ordenación por **inserción directa** es el que generalmente utilizan los jugadores de cartas cuando ordenan éstas, de ahí que también se conozca con el nombre de **método de la baraja**.

La idea central de este algoritmo consiste en insertar un elemento del arreglo en la parte izquierda del mismo, que ya se encuentra ordenada. Este proceso se repite desde el segundo hasta el n-ésimo elemento.

También se puede elaborar en una lista separada iniciando con un elemento que ya se encuentra ordenado. La lista ira creciendo de manera ordenada conforme se inserten nuevos elementos.

Método de Shell

Recibe este nombre en honor a su autor Donald L. Shell. Este método también se conoce con el nombre de inserción **con incrementos decrecientes**.

La idea general del algoritmo es la siguiente:

1. Se divide la lista original de n elementos en n/2 grupos de dos con un intervalo entre los elementos de cada grupo de n/2 y se clasifica cada grupo por separado; es decir, se comparan las parejas de elementos y si no están ordenados se intercambian ebtre sí de posiciones.

2. Se divide ahora la lista en n/4 grupos de cuatro con un intervalo de n/4 y se clasifican los datos de cada grupo.

3. Se repite el proceso hasta que, en el último paso se clasifica el grupo de n elementos.

Método Quicksort

El algoritmo básico fue inventado en 1960 y es por lo general el algoritmo de ordenación más rápido y eficiente. El método consiste en lo siguiente:

1. Se toma un elemento X de una posición cualquiera del arreglo.
2. Se trata de ubicar a X en la posición correcta del arreglo, de tal forma que todos los elementos que se encuntran a su izquierda sean menores o iguales a X y todos los elementos que se encuentarn a su derecha sean mayores o iguales a X.
3. Se repiten los pasos anteriores pero ahora para los conjuntos de datos que se encuntran a la izquierda y a la derecha de la posición correcta de X en el arreglo.
4. El proceso termina cuando todos los elementos se encuentran en su posición correcta en el arreglo.

Algoritmo:

```
 quickSort( arr[],  izq,  der)
{
   tipoDato  x, aux

      x ← arr[der];
      i ← izq;
      j ← der;
      Repite
        mientras arr[i]<x
          i ←i+1
        fin_mientras
        mientras arr[j]>x
          j ←j-1
        fin_mientras
        si i<=j
          aux ←arr[i]
          arr[i]←arr[j]
          arr[j]←aux
          i ←i+1
          j ←j-1
        fin_si
      Hasta que i>j

      si izq<j
        quickSort(arr, izq, j)
      fin_si
      si i<der
        quickSort(arr, i, der)
     fin_si
fin.
```

La llamada inicial sería *quicksort(arreglo, 1, n) ó quickSort(arreglo, 0, n-1)* en C.

MÉTODOS DE BÚSQUEDA

Buscar en una tabla un determinado valor significa localizar un elemento de la tabla cuyo contenido coincida con él. Si no existe ningún elemento coincidente con el valor buscado, se deberá indicar esta situación mediante un mensaje.

Es evidente que la búsqueda se puede realizar tanto en vectores ordenados como en vectores desordenados, utilizando diferentes algoritmos de búsqueda.

BÚSQUEDA EN UN VECTOR DESORDENADO. BÚSQUEDA SECUENCIAL.

Si la tabla está desordenada sólo hay una forma de búsqueda: la búsqueda secuencial.

El algoritmo consiste básicamente en recorrer la tabla comparando el valor que se desea localizar con cada uno de los elementos del mismo hasta que se encuentre. En ese momento se podrá saber, además, en qué posición está situado dicho elemento.

Si se recorre la tabla completa y no se encuentra, se emitirá un mensaje indicativo de su no existencia.

La búsqueda se realizará mediante un ciclo repetitivo, cuya condición de salida será que, **o bien se encuentre el elemento deseado, o bien se termine la tabla**. Al salir del ciclo, debido a que ha podido ser por dos motivos diferentes, se preguntará cuál de ellos fue y se procederá con la acción correspondiente.

El pseudocódigo de búsqueda secuencial para localizar un valor (que se introduce desde el teclado) en un vector desordenado V, de N elementos y que utiliza una variable BANDERA de tipo booleano, sería:

```
inicio
        Introducir VALOR
        i ← 1
        BANDERA ← FALSO
        mientras (i ≤ N)  y (BANDERA = FALSO)
                si V[i] = VALOR
                        entonces
                                BANDERA ← VERDADERO
                        si no
                                i ← i + 1
                fin_si
        fin_mientras
        si BANDERA = VERDADERO
                entonces
                        Imprimir "Valor encontrado en posición: ", i
                si no
                        Imprimir "Valor no encontrado"
        fin_si
fin
```

BÚSQUEDA EN UN VECTOR ORDENADO. BÚSQUEDA SECUENCIAL.

Este método es similar al visto anteriormente, con la diferencia de que si la tabla está ordenada de forma ascendente, se puede detectar que el valor buscado no está en ella, en el momento en que se encuentre un elemento con un valor superior a él.

La búsqueda de un elemento consiste en recorrer la tabla mediante un ciclo repetitivo, cuya **condición de salida** no **será** sólo que se encuentre el elemento, o que se termine la tabla, sino **también que se alcance un elemento con contenido mayor que el buscado**.

Al salir de este ciclo se averiguará si el motivo fue la localización o no del elemento, y se emitirá en este caso el correspondiente mensaje.

El pseudocódigo de búsqueda secuencial para localizar un valor en una tabla ordenada crecientemente V, de N elementos, sería:

```
inicio
        Introducir VALOR
        i ← 1
        BANDERA ← FALSO
        mientras (i ≤ N)  y (BANDERA = FALSO) y (VALOR ≥ V[i])
                si VALOR = V[i]
                        entonces
                                BANDERA ← VERDADERO
                        si no
                                i ← i + 1
                fin_si
        fin_mientras
        si BANDERA = VERDADERO
                entonces
                        Imprimir "Valor encontrado en posición: ", i
                si no
                        Imprimir "Valor no encontrado"
        fin_si
fin
```

Si la tabla estuviese ordenada descendentemente, se sabría que el valor buscado no está en ella cuando, al compararlo con un elemento de la tabla, éste resultase ser menor.

BÚSQUEDA EN UN VECTOR ORDENADO. BÚSQUEDA BINARIA Ó DICOTÓMICA.

La búsqueda secuencial **NO es el mejor método de búsqueda si el arreglo tiene una gran número de elementos**, ya que puede resultar muy lento. Es más rápido y, por lo tanto, más eficaz, la búsqueda binaria, también llamada **dicotómica**. Este método es el que habitualmente utilizamos los humanos para localizar un determinado concepto en una lista ordenada, por ejemplo, un directorio telefónico o un diccionario.

Este método consiste en lo siguiente:

1. Se compara el valor a localizar con el del elemento central del arreglo:

- Si coinciden, se ha encontrado y, por lo tanto, se ha terminado la búsqueda.
- Si no coinciden, se determina si el valor buscado debe estar en la mitad izquierda o derecha del arreglo, dependiendo de si es inferior o superior, respectivamente, del elemento central.

La posición del elemento central del arreglo se calcula mediante la fórmula:

POS_MED = ENT ((POS_INI + POS_FIN) / 2)

Donde POS_INI y POS_FIN son las posiciones inicial y final del subarreglo que se está considerando.

1. En la mitad donde se deba continuar la búsqueda, se procede de la misma manera, es decir, se compara el valor buscado con el elemento central de esa mitad.

- Si coincide, se ha localizado.
- Si no coincide, se determina en cuál de las dos mitades de esta submitad debe estar el valor deseado.

1. Se sigue procediendo de este modo (dividiendo el arreglo en submitades) hasta que la búsqueda:

- Se termine con éxito (se encontró el elemento).
- Se termine sin éxito (no se pudo encontrar el elemento por no existir en el arreglo).

Se considerará que el valor buscado no existe en el arreglo cuando POS_INI y POS_FIN coincidan y su contenido sea diferente de dicho valor.

En otras palabras, se puede decir que la búsqueda binaria de un determinado valor consiste en ir eliminando las mitades donde se tiene la certeza de que dicho valor no se encuentra.

El pseudocódigo de búsqueda binaria para localizar un valor en un arreglo ordenado ascendentemente V, de N elementos, sería:

```
inicio
    Introducir VALOR
    POS_INI ← 1
    POS_FIN ← N
    POS_MED ← ENT( (POS_INI + POS_FIN) / 2)
    mientras (VALOR ≠ V[POS_MED]) y POS_FIN > POS_INI)
        si VALOR > V[POS_MED]
            entonces
                POS_INI ← POS_MED + 1
            si no
                POS_FIN ← POS_MED – 1
        fin_si
        POS_MED ← ENT( (POS_INI + POS_FIN) / 2)
    fin_mientras
    si VALOR = V[POS_MED]
        entonces
            Imprimir “Posición: “, POS_MED
        si no
            Imprimir “No existe en el arreglo.”
    fin_si
fin
```

Búsqueda por transformación de claves (Hash)

Este método, llamado **por transformación de claves**, permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados. Cuenta también con la ventaja de que **el tiempo de búsqueda es** prácticamente **independiente del número de componentes del arreglo**.

Este método trabaja basándose en una **función de transformación** o función hash (h) que convierte una clave dada en una dirección (índice) dentro del arreglo.

dirección ← H(clave)

La función hash es aplicada a la clave da un índice del arreglo, lo que permite accesar **directamente** sus elementos.

Siempre debe equilibrarse el costo por espacio de memoria con el costo por tiempo de búsqueda.

Cuando se tienen claves que no se corresponden con índices (por ejemplo, por ser alfanuméricas), o bien cuando las claves son valores numéricos muy grandes, debe utilizarse una función hash que permita transformar la clave para obtener una dirección apropiada.

Esta función hash debe ser simple de calcular y debe asignar direcciones de la manera más uniforme posible. Es decir, dadas dos claves diferentes debe generar posiciones diferentes. Si esto no ocurre **($H(K_1) = d$, $H(K_2) = d$ y $K_1 \neq K_2$)**, hay una colisión.

Se define, entonces, una **colisión** como la asignación de una misma dirección a dos o más claves distintas.

Para trabajar con este método de búsqueda debe elegirse previamente:

- Una función hash que sea fácil de calcular y que distribuya uniformemente las claves.
- Un método para resolver colisiones. Si éstas se presentan se debe contar con algún método que genere posiciones alternativas.

Funciones hash

No hay reglas que permitan determinar cuál será la función más apropiada para un conjunto de claves, de tal manera que asegure la máxima uniformidad en la distribución de las mismas.

1. Función módulo (por división)

Consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo. Supóngase que se tiene un arreglo de N elementos, ya sea K la clave del dato a buscar. La función hash queda definida por la siguiente fórmula:

$$H(k) = (K \bmod N) + 1$$

En la fórmula puede observarse que al residuo de la división se le suma 1, esto es para obtener un valor entre 1 y N.

Para lograr una mayor uniformidad en la distribución, N debe ser un número primo o divisible por muy pocos números. Por lo tanto, dado N, si éste no es un número primo se tomará el valor primo más cercano.

Ejemplo:

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K_1 = 7259 y K_2 = 9359 dos claves a las que deban asignarse posiciones en el arreglo. Se aplica la fórmula con N = 100, para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1) = (7259 \bmod 100) + 1 = 60$

$H(K_2) = (9359 \bmod 100) + 1 = 60$

Como $H(K_1) = H(K_2)$ y $K_1 \neq K_2$, se está ante una colisión.

Se aplica ahora la fórmula con N igual a un valor primo en vez de utilizar N = 100.

$H(K_1) = (7259 \bmod 97) + 1 = 82$

$H(K_2) = (9359 \bmod 97) + 1 = 48$

Con N = 97 se ha eliminado la colisión.

1. Función cuadrado

Consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección. El número de dígitos a tomar queda determinado por el rango del índice. Sea K la clave del dato a buscar. La función hash queda definida por la siguiente fórmula:

$$\mathbf{H}(K) = \text{dígitos\_centrales}\ (K^2) + 1$$

La suma de una unidad a los dígitos centrales es para obtener un valor entre 1 y N.

Ejemplo:

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K_1 = 7259 y K_2 = 9359 dos claves a las que deban asignarse posiciones en el arreglo. Se aplica la fórmula para calcular las direcciones correspondientes a K_1 y K_2.

$K_1^2 = 52\ 693\ 081$

$K_2^2 = 87\ 590\ 881$

$H(K_1)$ = dígitos_centrales (52 693 081) + 1 = 94

$H(K_2)$ = dígitos_centrales (87 590 881) + 1 = 91

Como el rango de índices en el arreglo varía de 1 a 100 se toman solamente los dos dígitos centrales del cuadrado de las claves.

1. Función plegamiento

Consiste en dividir la clave en partes de igual número de dígitos (la última puede tener menos dígitos) y operar con ellas, tomando como dirección los dígitos menos significativos. La operación entre las partes puede hacerse por medio de sumas o multiplicaciones. Sea K la clave del dato a buscar. K está formada por los dígitos $d_1, d_2, \ldots, d_n$. La función hash queda definida por la siguiente fórmula:

$$H(K) = \text{dígmensig } ((d_1 \ldots d_i) + (d_i \ldots d_j) + \ldots + (d_1 \ldots d_n)) + 1$$

ó

$$H(K) = \text{dígmensig } ((d_1 \ldots d_i) * (d_i \ldots d_j) * \ldots * (d_1 \ldots d_n)) + 1$$

El operador que aparece en la fórmula operando las partes de la clave es el de suma. La suma de una unidad a los dígitos menos significativos (dígmensig) es para obtener un valor entre 1 y N.

Ejemplo:

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K_1 = 7259 y K_2 = 9359 dos claves a las que deban asignarse posiciones en el arreglo. Se aplica la fórmula para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1)$ = dígmensig (72 + 59) + 1 = dígmensig (131) + 1 = 32

$H(K_2)$ = dígmensig (93 + 59) + 1 = dígmensig (152) + 1 = 53

De la suma de las partes se toman solamente dos dígitos porque los índices del arreglo varían de 1 a 100.

1. Función truncamiento

Consiste en tomar algunos dígitos de la clave y formar con ellos una dirección. Este método es de los más sencillos, pero es también de los que ofrecen **menos uniformidad en la distribución de las claves**.

Sea K la clave del dato a buscar. K está formada por los dígitos dígitos $d_1, d_2, \ldots, d_n$. La función hash queda definida por la siguiente fórmula:

$$H(K) = \text{elegirdígitos } (d_1, d_2, \ldots, d_n.) + 1$$

La elección de los dígitos es arbitraria. Podrían tomarse los dígitos de las posiciones impares o de las pares. Luego podría unírseles de izquierda a derecha o de derecha a izquierda. La suma de una unidad a los dígitos seleccionados es para obtener un valor entre 1 y 100.

Ejemplo:

Sean N = 100 el tamaño del arreglo, y sean sus direcciones los números entre 1 y 100. Sean K_1 = 7259 y K_2 = 9359 dos claves a las que deban asignarse posiciones en el arreglo. Se aplica la fórmula para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1)$ = elegirdígitos (7 2 5 9) + 1 = 76

$H(K_2)$ = elegirdígitos (9 3 5 9) + 1 = 96

En este ejemplo se toma el primer y tercer número de la clave y se une éste de izquierda a derecha.

En todos los casos anteriores se presentan ejemplos de claves numéricas. Sin embargo, en la realidad las claves pueden ser **alfabéticas** o **alfanuméricas**. En general, cuando aparecen letras en las claves se suele asociar a cada una un entero a efectos de convertirlas en numéricas.

Si la clave fuera **ADA**, su equivalente numérico sería 010401. Si hubiera combinación de letras y números, se procedería de la misma manera. Por ejemplo, para la clave **Z4F21**, su equivalente numérico sería 2740621.

Otra alternativa sería, para cada carácter, tomar su valor decimal asociado según el código ASCII. Una vez obtenida la clave en su forma numérica, se puede utilizar normalmente cualquiera de las funciones arriba mencionadas.

Solución de colisiones.

La elección de método adecuado para resolver colisiones es tan importante como la elección de una buena función hash. Normalmente, cualquiera que sea el método elegido, resulta costoso tratar las colisiones. Es por ello que debe hacerse un esfuerzo por encontrar la función que ofrezca mayor uniformidad en la distribución de las claves.

La manera más natural de resolver el problema de las colisiones es reservar una casilla por clave. Es decir, que aquellas se correspondan una a una con las posiciones del arreglo. Pero como ya se mencionó, esta solución puede tener un alto costo en memoria. Por lo tanto deben analizarse otras alternativas que permitan equilibrar el uso de memoria con el tiempo de búsqueda.

Reasignación

Existen varios métodos que trabajan bajo el principio de comparación y reasignación de elementos. Se analizarán tres de ellos:

- Prueba lineal
- Prueba cuadrática
- Doble dirección hash

a) Prueba lineal

Consiste en que una vez detectada la colisión **se debe recorrer el arreglo secuencialmente a partir del punto de colisión, buscando al elemento**. El proceso de búsqueda concluye cuando el elemento es hallado, o bien cuando se encuentra una **posición vacía**. Se trata al arreglo como a una **estructura circular**: el siguiente elemento después del último es el primero.

```
PRUEBALINEAL (V, N, K)
Inicio
D ← H(K) {Genera dirección}
Si V[D] = K
   entonces
      Escribir "El elemento está en la posición", D
   si no
      DX ← D + 1
      mientras (DX ≤ N) y (V[DX] ≠ K) y (V[DX] ≠ VACÍO) y (DX ≠ D)
         DX ← DX + 1
         Si DX = N + 1
            entonces
               DX ← 1
         fin_si
      fin_mientras
      Si V[DX] = K
         entonces
            Escribir "El elemento está en la posición", DX
         si no
            Escribir "El elemento no está en el arreglo"
               fin_si
fin_si
Fin
```

La cuarta condición del ciclo mientras (**DX ≠ D**), es para evitar caer en un **ciclo infinito**, si el arreglo estuviera lleno y el elemento a buscar no se encontrara en él.

La principal desventaja de este método es que puede haber un **fuerte agrupamiento alrededor de ciertas claves**, mientras que otras zonas del arreglo permanecerían vacías. Si las concentraciones de claves son muy frecuentes, la **búsqueda será** principalmente **secuencial** perdiendo así las ventajas del método hash.

a) Prueba cuadrática

Este método es similar al de la prueba lineal. La diferencia consiste en que en el cuadrático las direcciones alternativas se generarán como D + 1, D + 4, D + 9, …, D + i^2 en vez de D + 1, D + 2, …, D + i. Esta variación permite una **mejor distribución de las claves colisionadas**.

```
PRUEBACUADRÁTICA (V, N, K)
Inicio
        D ← H(K) {Genera dirección}
   Si V[D] = K
      entonces
         Escribir "El elemento está en la posición", D
      si no
         I ← 1
         DX ← D + I^2
         Mientras (V[DX] ≠ K) y (V[DX] ≠ VACÍO)
               I ← I + 1
            DX ← D + I^2
            Si DX > N
               entonces
                  I ← 0
                  DX ← 1
                  D ← 1
                           fin_si
                     fin_mientras
                     Si V[DX] = K
                           entonces
                                 Escribir "El elemento está en la posición", DX
                           si no
                                 Escribir "El elemento no está en el arreglo"
                     fin_si
   fin_si
Fin
```

La principal desventaja de este método es que **pueden quedar casillas del arreglo sin visitar**. Además, como los valores de las direcciones varían en I^2 unidades, **resulta difícil determinar una condición general para detener el ciclo mientras**. Este problema podría solucionarse empleando una variable auxiliar, cuyos valores dirijan el recorrido del arreglo de tal manera que garantice que serán visitadas todas las casillas.

a) Doble dirección hash

Consiste en que una vez detectada la colisión se debe generar otra dirección aplicando la función hash a la dirección previamente obtenida. El proceso se detiene cuando el elemento es hallado, o bien cuando se encuentra una posición vacía.

```
D       H(K)
D'      H(D)
```

D'' H(D')
…

La función hash que se aplique a las direcciones puede o no ser la misma que originalmente se aplicó a la clave. No existe una regla que permita decidir cuál será la mejor función a emplear en el cálculo de las sucesivas direcciones.

```
DOBLEDIRECCIÓN (V, N, K)
Inicio
        D ← H(K) {Genera dirección}
   Si V[D] = K
      entonces
         Escribir "El elemento está en la posición", D
      si no
         DX ← H'(D)
         Mientras (DX ≤ N) y (V[DX] ≠ K) y (V[DX] ≠ VACÍO) y (DX ≠ D)
            DX ← H'(DX)
                        fin_mientras
                        Si V[DX] = K
                                entonces
                                        Escribir "El elemento está en la posición", DX
                                si no
                                        Escribir "El elemento no está en el arreglo"
                        fin_si
   fin_si
Fin
```

d) Arreglos anidados

Este método consiste en que cada elemento del arreglo tenga otro arreglo en el cual se almacenen los elementos colisionados. Si bien la solución parece ser sencilla, es claro también que resulta ineficiente. Al trabajar con arreglos se depende del espacio que se haya asignado a éste. Lo cual conduce a un nuevo problema difícil de solucionar: elegir un tamaño adecuado de arreglo que permita un equilibrio entre el costo de memoria y el número de valores colisionados que pudiera almacenar.

e) Encadenamiento

Consiste en que cada elemento del arreglo tenga un apuntador a una lista ligada, la cual se irá generando e irá almacenando los valores colisionados a medida que se requiera. Es el método más eficiente debido al dinamismo propio de las listas. Cualquiera que sea el número de colisiones registradas en una posición, siempre será posible tratar una más. Como desventajas del método de encadenamiento se cita el hecho de que ocupa espacio adicional al de la tabla, y que exige el manejo de listas ligadas. Además, si las listas crecen demasiado se perderá la facilidad de acceso directo del método hash.

Ejercicios

1. Escriba un programa para búsqueda secuencial en un arreglo desordenado, que obtenga todas las ocurrencias de un dato dado.

1. Dado un arreglo que contiene los nombres de N alumnos ordenados alfabéticamente, escriba un programa que encuentre un nombre dado en el arreglo. Si lo encuentra debe regresar como resultado la posición en la que lo encontró. En caso contrario, debe enviar un mensaje adecuado.

1. Dado un arreglo de N componentes que contienen la siguiente información:

   - Nombre del alumno
   - Promedio
   - Número de materias aprobadas

Escriba un programa que lea el nombre de un alumno y regrese como resultado el promedio y el número de materias aprobadas por dicho alumno. Si el nombre dado no está en el arreglo, envié un mensaje adecuado.

   a) Considere que el arreglo está desordenado
   b) Considere que el arreglo está ordenado

1. Escriba un programa para búsqueda secuencial en arreglos ordenados de manera descendente.

1. Escriba un programa para búsqueda secuencial en listas enlazadas desordenadas. Si el elemento se encuentre en la lista, indique el número de nodo en el cual se encontró. En caso contrario, emita un mensaje adecuado.

1. Escriba un programa para búsqueda secuencial en listas enlazadas ordenadas de manera descendente.

1. Escriba un programa de búsqueda binaria en arreglos ordenados.

   a) De manera ascendente
   b) De manera descendente

1. Resuelva el inciso b del problema 3 utilizando el algoritmo de búsqueda binaria.

1. Dado que se quiere almacenar los registros con claves: 23, 42, 5, 66, 14, 43, 59, 81, 37, 49, 28, 55, 94, 80 y 64 en un arreglo de 20 elementos, defina una función hash que distribuya los registros en el arreglo. Si hubiera colisiones, resuélvalas aplicando el método de reasignación lineal.

1. De un grupo de N alumnos se tienen los siguientes datos:

- Matrícula: valor entero comprendido entre 1000 y 4999
- Nombre: cadena de caracteres
- Dirección: cadena de caracteres

El campo clave es matrícula.

Los N registros han sido almacenados en un arreglo, aplicando la siguiente función hash:

$$H(clave) = dígitos\_centrales\ (clave^2) + 1$$

Las colisiones han sido tratadas con el método de doble dirección hash.

Escriba un subprograma que lea la matrícula de un alumno y regrese como resultado el nombre y dirección del mismo. En caso de no encontrarlo emita un mensaje adecuado.

1. Se quiere almacenar en un arreglo los siguientes datos de N personas:

- Clave de contribuyente: alfanumérico, de longitud 6.
- Nombre: cadena de caracteres
- Dirección: cadena de caracteres
- Saldo: real

Defina una función hash para almacenar en un arreglo los datos mencionados. Utilice el método de encadenamiento para resolver las colisiones.

1. Presente y explique una función hash que permita almacenar en un arreglo los elementos de la tabla periódica de los elementos químicos y sus propiedades, de una manera uniforme. La clave está dada por el nombre de los elementos.

1. Utilice la función definida en el ejercicio anterior para insertar y eliminar los elementos que se dan a continuación:

Insertar: sodio, oro, osmio, litio, boro, cobre, plata, radio.
Eliminar: oro, osmio, boro, cobre, plata.

1. Dados los 12 signos del zodiaco (capricornio, acuario, piscis, aries, tauro, géminis, cáncer, leo, virgo, libra, escorpión, sagitario).

a) Escriba un subprograma para almacenarlos en una estructura tries.
b) Escriba un subprograma de búsqueda para los signos, almacenados según lo especificado en el inciso anterior.