Registro con variantes

Un registro es una asociación de datos de tipos arbitrarios, estructurados o no, en uno compuesto, creando una estructura de datos formada por una colección de elementos llamados campos. Se define de la forma:

TYPE T = RECORD s_1 : T_1
s_2 : T_2
… …
s_n : T_n

Los identificadores $s_1,s_2,\ldots,s_n$ son los nombres de las componentes individuales de las variables de ese tipo. El i-ésimo campo de la variable X : T se simboliza por $x.s_i$. Hay aplicaciones en las que el tipo de dato varía según las circunstancias. Se introduce una tercera componente, el tipo discriminador o campo objetivo. Esta situación da lugar a la estructura del registro con variante. Su forma general para m variantes es:

TYPE T = RECORD s_1 : T_1; s_2 : T_2; … ; s_{n-1} : T_{n-1}
CASE s_n : T_n OF
v_1:s_{11}:T_{11} ; … ; $s_{1,n1}$: $T_{1,n1}$|
v_2:s_{21}:T_{21} ; … ; $s_{2,n2}$: $T_{2,n2}$|
… … …
v_m:s_{m1}:T_{m1} ; … ; $s_{m,n\,m}$: $T_{m,n\,m}$|
END;
END;

Los elementos $s_1,s_2,\ldots,s_{n-1}$ son los identificadores de campo que pertenecen a la parte común del registro, $s_{i,j}$ son los nombres del selector de los campos que pertenecen a la parte variante y s_n es el campo objetivo discriminante de tipo T_n. Las constantes $v_1,v_2, \ldots ,v_n$ denotan los valores fijos del campo objetivo.

Para evitar el uso de selectores de componentes erróneos, las operaciones con las variantes individuales se agrupan en una proposición CASE:

CASE X.s_n OF
v_1:s_1|
v_2:s_2|
…
v_m:s_m
END;

Donde S_k representa el número de proposiciones en el caso que X tome la forma variante k.

Aplicación del registro con variantes

El procedimiento calcula la distancia d entre dos puntos A y B dadas las variables a, b del tipo registro variante coordenada. El cálculo difiere según las cuatro combinaciones de coordenadas cartesianas y polares. La definición de tipos es:

```
TYPE CordMode = (Cartesiana, Polar)
TYPE Coordenada = RECORD
                    CASE Clase : CordMode OF
                           Cartesiana : x,y : REAL |
                           Polar : r, phi : REAL
                    END;
                  END
```

El nombre objetivo es **Clase** y los nombre de las coordenadas son x, y en el caso cartesiano y r, phi en el caso polar. El cálculo se realiza:

```
CASE a.Clase OF
   Cartesiana : CASE b.Clase OF
     Cartesiana: d:= sqrt (sqr (a.x - b.x) + sqr (a.y - b.y))
     Polar : d:= sqrt (sqr (a.x - b.r*cos(b.phi) + (sqr (a.y - b.r*sin(b.phi)
   Polar : CASE b.Clase OF
     Cartesiana: d:= sqrt (sqr (a.r*cos(a.phi) - b.x) + sqr (a.r*sin(a.phi) - b.y))
     Polar : d:= sqrt (sqr (a.r*cos(a.phi) - b.r*cos(b.phi) + (sqr (a.r*sin(a.phi) -b.r*sin(b.phi))
```

```
END; END;
```

En el caso de utilizarse dos registros sin variantes cada uno de ellos representaría un tipo de coordenada, tendríamos:

```
TYPE Cartesiana = RECORD
                  x,y : REAL
                  END
TYPE Polar = RECORD
                  r, phi : REAL
                  END
```

Necesitaríamos dos estructuras de datos definidas que deberían estar mapeadas sobre cuatro variables:

```
VAR   A_Cartesiana: Cartesiana,
      B_Cartesiana: Cartesiana,
      A_Polar : Polar,
      B_Polar : Polar;
```

Además debemos disponer de algún indicador para saber si el punto A viene dado en coordenadas cartesianas o polares y otro idéntico para el punto B, con lo que tenemos dos variables más definidas:

```
VAR A_Es_Cartesiana, B_Es_Cartesiana : BOOLEAN;
```

Con lo que el procedimiento de cálculo quedaría de la siguiente forma:

```
IF A_Es_cartesiana THEN
  IF B_Es_Cartesiana THEN
        d:= sqrt (sqr (a.x - b.x) + sqr (a.y - b.y))
  ELSE
        d:= sqrt (sqr (a.x - b.r*cos(b.phi) + (sqr (a.y - b.r*sin(b.phi))
  END
ELSE
  IF B_Es_Cartesiana THEN
        d:= sqrt (sqr (a.r*cos(a.phi) - b.x) + sqr (a.r*sin(a.phi) - b.y))
  ELSE
        d:= sqrt (sqr (a.r*cos(a.phi) - b.r*cos(b.phi) + (sqr (a.r*sin(a.phi) -b.r*sin(b.phi))
  END
END
```

Es evidente que con este sistema el empleo de memoria es mayor y el algoritmo empleado es menos legible.

Búsqueda KMP

Este algoritmo de búsqueda se basa en analizar la información de las comparaciones realizadas entre el texto y el patrón antes de la primera inconcordancia, con el fin de lograr un avance más rápido en la cadena de texto que en la búsqueda directa. Para ello se realiza la precompilación del patrón, que obtiene el número de caracteres iguales con respecto al primer carácter. De forma que cada vez que dos caracteres comparados no concuerdan el modelo cambiará totalmente.

Precompilación

```
j:= 0 ; k:= -1 ; d[0] := -1
WHILE (j<M-1) DO
        WHILE (k>=0) AND (P[j]#P[k]) DO k:=d[k]; END;
        j := j+1 ; k:= k+1
        IF P[j] = P[k] THEN d[j]:=d[k]  ELSE d[j] := k END;
END;
```

Búsqueda

```
i:=0; j:=0;
WHILE (j<M) AND (i<N) DO
        WHILE (j>0) AND (s[i]#P[j] DO j:=d[j] END;
        i := i+1 ; j:= j+1
END
IF j=M THEN WriteString ("Encontrado")
```

Ejemplo

Hoola-Hoola girls like Hooligans
Hooligan
Hooligan
Hooligan
…..
Hooligan

El número de comparaciones es de N + M, siendo M el número de elementos del patrón y N el de la cadena.

Búsqueda BM

El algoritmo de KMP se basa en las concordancias del patrón, lo que es poco frecuente y el avance en la búsqueda rara vez es mayor que uno. El algoritmo BM mejora este rendimiento.

La precompilación del patrón se realiza asignando a un arreglo de 128 elementos el tamaño del patrón. Luego al número correspondiente en el código ASCII del carácter que analizamos se le asigna la distancia hasta el final del patrón.

Este algoritmo empieza las comparaciones desde el final del patrón ha buscar.

Precompilación

```
FOR ch:=0C TO 177C DO d[ch]:=M END;
FOR j:=0 TO M-2 DO d[P[j]]:=M-j-1 END;
```

Búsqueda

```
i:=M
REPEAT
        j:=M; k:=i;
        REPEAT k:=k-1; j:=j-1
        UNTIL (j<0) OR (P[j]#s[k])
        i:= i+d[s(i-1)]
UNTIL (j<0) OR (i>=N)
IF (j<0) THEN WriteString("Hallado")
```

Ejemplo

Hoola-Hoola girls like Hooligans
Hooligan
Hooligan
…..
Hooligan

El número de comparaciones es en el mejor caso de N/M y en el caso medio de N

Inserción directa

En cada paso, comenzando con i=2 y aumentando en una unidad, el i-ésimo elemento de la secuencia fuente se toma y se transfiere a la secuencia destino insertándolo en el lugar correspondiente. Se aplica la técnica del centinela como condición de terminación. El algoritmo completo es:

```
PROCEDURE StraightInsertion;
        VAR i,j: index; x: item;
BEGIN
        FOR i:=2 TO n DO
                a[0]:=a[i]; j:=i;
                WHILE a[0]<a[j-1] DO a[j]:=a[j-1]; j:=j-1 END;
                a[j]:=a[0]
        END
END StraightInsertion
```

Análisis:

El número de comparaciones C en la i-ésima extracción es a lo sumo i-1 y como mínimo 1; suponiendo que todas las permutaciones de n llaves sean igualmente probables (i/2 en promedio). El número M de movimientos (asignación de elementos) es C_i+2 (incluido el centinela). Por tanto, los números totales de comparaciones y movimientos son:

Comparaciones	*Movimientos*
$C_{min} = n-1$	$M_{min} = 3 * (n-1)$
$C_{prom} = (n^2+n-2) / 4$	$M_{prom} = (n^2+ 9n-10)/4$
$C_{max} = (n^2+n-2)/2$	$M_{max} = (n^2+ 5n -4)/2$

Los números mínimos corresponden al caso en que los elementos se hallan en orden al inicio; el peor caso se presenta cuando los elementos se hallan en orden inverso, es decir un comportamiento natural.

Inserción binaria

Este algoritmo se basa en el de inserción directa teniendo en cuenta que la secuencia destino donde debe insertarse el elemento ya esta ordenada. Partiendo de esta premisa, la elección obvia es una búsqueda binaria que prueba la secuencia destino en la mitad y continúa buscando hasta encontrar el punto de inserción. El algoritmo es:

```
PROCEDURE BinaryInsertion;
        VAR i,j,m,L,R : index; x:item;
BEGIN
        FOR i:=2 TO n DO
                x:=a[i] ; L:= 1 ; R:=i;
                WHILE L < R DO
                        m:=(L+R) DIV 2
                        IF a[m] <= x THEN L := m+1
                        ELSE R := m
                        END;
                END;
                FOR j:=i TO R+1 BY -1 DO
                        a[j] := a[j-1]
                END;
                a[R] := x
        END;
END BinaryInsertion.
```

Análisis:

La posición de inserción se encuentra cuando L=R. En consecuencia, el intervalo de búsqueda debe ser, al final, de longitud 1; lo que supone dividir a la mitad el intervalo de longitud i **logi** veces. Por tanto:

$$C = \sum i : 1 \le i \le n : \lceil \log i \rceil$$

Suma que aproximada por la integral queda de forma:

n*(logn-c)+c

donde c=log e = 1/ln 2=1.44269… El número de comparaciones es esencialmente independiente del orden inicial de los elementos. Sin embargo dado el valor truncador de la división el número de comparaciones puede ser hasta uno más de lo previsto. Esto es debido a que las posiciones de inserción en el extremo inferior se localizan en promedio más pronto que las del extremo superior, con lo que se favorecen los casos en que los elementos están fuera de orden al principio. El número menor corresponde a cuando es orden es inverso y el máximo cuando ya están en orden.

La mejora sobre la inserción directa es sólo en el número de comparaciones, no al de movimientos requeridos. Como mover elementos es más lento que compararlos la mejora no es radical.

El orden del algoritmo sigue siendo de n^2. La clasificación por inserción no es un método muy adecuado para computadoras.

Clasificación por selección directa

Se basa en los siguientes principios:

1- Seleccionar el elemento con llave menor.
2- Intercambiarlo con el primer elemento.
3- Repetir las operaciones con n-1, luego con n-2 hasta que sólo quede un elemento.

```
PROCEDURE StraightSelection:
        VAR i,j,k:index; x:item;
BEGIN
        FOR i:=1 TO n-1 DO
          k:=i; x:=a[i];
          FOR j:=i+1 TO n DO
            IF a[j]<x THEN k:=j; x:=a[k] END
          END;
          a[k]:=a[i]; a[i]:=x
        END
END StraightSelection
```

Análisis:

El número de comparaciones es independiente del orden inicial de llaves n-1, n-2,…,1, es decir, una progresión aritmética y obtenemos:

$C=(n^2+n-2)/2$

El número de movimientos es por lo menos, en el caso de llaves iniciales ordenadas:

$M_{min}= 3*(n-1)$

y como máximo, si al inicio están en orden inverso:

$M_{max}= (n^2+n-2)/2+3(n-1)$

Como promedio, dado que el procedimiento rastrea el arreglo, comparando cada elemento con el valor mínimo descubierto hasta el momento y, si es más pequeño hace una asignación.. La probabilidad de que el segundo elemento sea menor que el primero es ½, la del tercero 1/3, la del cuarto ¼, así sucesivamente, con lo que el número total esperado de movimiento es H_{n-1}, donde H_n es el enésimo número armónico

$H_n= 1 + ½ + 1/3 + … + 1/n$

que se puede expresar como:

$H_n= \ln n + g + ½ n - 1/12n^2+ …$

donde g es la constante de Euler. Con una n suficientemente grande se puede aproximar el número promedio de asignaciones como:

$F_i= \ln i + g + l$

El número promedio de movimientos en la clasificación por selección, es pues, la suma de F_i con i desde 1 hasta n, aproximando la integral se obtiene el valor aproximado:

$M_{prom}= n(\ln (n) + g)$

Clasificación Shell

Es un refinamiento del método de inserción directa, se basa en la comparación de grupos de elementos separados por una distancia h, que se reduce en cada pase. Cada uno de los grupos se ordena por inserción directa y se refunde junto a otro(s) en un nuevo grupo, que volverá a ser ordenado por inserción directa. De esta forma cada paso de clasificación incluye pocos elementos o ya están ordenados y se requieren pocos movimientos.

Cualquier secuencia de incremento es aceptable siempre que el último sea la unidad. Los incrementos no deben ser múltiplos de si mismos. Interesa que la interacción entre varias cadenas sea lo más a menudo posible; se cumple el siguiente teorema: si una secuencia clasificada con k se clasifica con i, debe permanecer clasificada con k. Dos secuencias que aporta Knuth son:

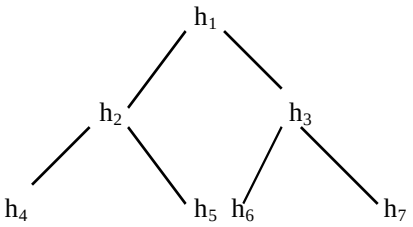
1, 4, 13, 40, 121
1, 3, 7, 15,31

con esta segunda secuencia, el análisis matemático produce un esfuerzo proporcional a $n^{2.1}$, necesario para clasificar n elementos.

```
CONST t=4
BEGIN
  h[1]:=9; h[2]:=5; h[3]:=3; h[4]:=1;
  FOR m:= 1 TO t DO
        k:= h[m]; s := -k
        FOR i:=k+1 TO n DO
              x:= a[i]; j:=i-k;
              IF s=0 THEN s:=-k END;
              s:=s+1; a[s]:=x;
              WHILE x<a[j] DO
                    a[j+k]:=a[j];
                    j:=j-k
              END;
              a[j+k]:=x
        END;
  END;
```

Clasificación por montón

Williams define un montón según la secuencia de llaves h_L, h_{L+1}, …, h_R, tales que $h_i \leq h_{2i}$ y $h_i \leq h_{2i+1}$ para i = L … R/2. De esta forma un árbol representado como:



forma un montón, donde h_1 es el elemento más pequeño. Si tenemos un montón h_{L+1}, …, h_R y debemos incluir un elemento, para formar el montón ampliado h_L, …, h_R ,se coloca el elemento en la cima y se deja que se hunda en la trayectoria de los elementos menores.

Floyd considera que en un arreglo h_1, …, h_n ; los elementos de la segunda mitad del arreglo forman un montón por si mismos, ya que en los elementos h_m, …, h_n con m= (n DIV 2) + 1, no hay dos índices i, j tales que j = 2 * i . A estos elementos se les puede considerar como el renglón de la parte inferior del árbol, entre los que no se requiere ninguna relación de ordenación. El montón debe ampliarse hacia la izquierda, con lo que en cada paso se incluye un nuevo elemento y se coloca correctamente con un desplazamiento.

Construcción de un montón:

44	55	12	42	94	18	06	67
44	55	12	67	94	18	06	42
44	55	18	67	94	12	06	42
44	94	18	67	55	12	06	42
94	67	18	44	55	12	06	42

Clasificación

94	67	18	44	55	12	06	42	
67	55	18	44	42	12	06	94	
55	44	18	06	42	12	67	94	
	44	42	18	06	12	55	67	94

42	12	18	06	44	55	67	94
18	12	06	42	44	55	67	94
12	06	18	42	44	55	67	94
12	06	18	42	44	55	67	94
06	12	18	42	44	55	67	94

El procedimiento completo de ordenación por montón llamado HeapSort es:

```
PROCEDURE HeapSort;
  VAR L, R index; x: item;
        PROCEDURE sift (L, R : index);
          VAR i,j : index; x:item;
        BEGIN
          i:=L ; j:=2*L; x:=a[L]
          IF (j<R) & (a[j]<a[j+1]) THEN j:=j+1 END;          Desplazamiento
          WHILE (j<=R) & (x<a[j]) DO
                a[i]:=a[j]; a[j] := x; i:= j ; j:= 2*j;
                 IF (j<R) & (a[j]<a[j+1]) THEN j:=j+1 END;
          END
        END sift;
BEGIN
  L:= (n DIV 2) +1 ; R:=n;                                  Genera un montón
  WHILE L>1 DO L:= L-1; sift (L,R) END;                     h1,…,hn
  WHILE R>1 DO
        x:=a[1]; a[1]:=a[R]; a[R]:=x;                       n pasos de
        R:=R-1; sift (L,R)                                  desplazamiento
  END
END HeapSort.
```

Análisis de clasificación por montón:

Aunque a primera vista no parezca que tiene buenos resultados ya que los números grandes son desplazados hacia la izquierda antes de depositarlos en el extremo derecho, esto produce que con pocos números sea poco eficiente, sin embargo con n suficientemente grande el método es muy eficiente y la eficiencia crece al crecer n.

En el peor caso, hay n/2 pasos de desplazamiento necesarios, pasando los elementos por log(n/2), log(n/2-1), …, log(n-1) posiciones, donde el logaritmo queda truncado al entero más bajo. Después, la fase de clasificación hace n-1 desplazamientos, con un máximo de log(n-1), log(n-2), …, 1 movimientos. Además hay n-1 movimientos para guardar a la derecha el elemento desplazado. Este argumento muestra que la clasificación por montón toma el orden de n*log(n) pasos aún en el peor caso. Si el orden inicial es el inverso, no se requieren movimientos en la creación del montón. El número promedio de movimientos es aproximadamente n/2*log(n) y con desviaciones pequeñas.

Clasificación por partición

Este procedimiento, inventado por Hoare, recibe el nombre de Quick Sort. Es el mejor algoritmo de ordenación. Está basado en el método de intercambio (burbuja), selecciona un elemento al azar (x), llamado comparando, se analiza el arreglo desde la izquierda hasta encontrar un elemento $a_i>x$, después se analiza desde la derecha hasta encontrar $a_j<x$, se intercambian ambos elementos y se prosigue hasta que los dos rastreos se encuentren. El arreglo queda dividido en dos partes, una con todos los elementos mayores o iguales al comparando y otra con todos los elementos menores o iguales. El proceso se repite en cada parte hasta que el arreglo quede ordenado.

```
PROCEDURE Sort(L,R:index)
  VAR i,j : index; w,x:item;
BEGIN
  i:=L; j:=R;
  REPEAT
        WHILE a[i]< x DO i := i+1 END;
        WHILE a[i]> x DO j := j-1 END;
        IF i <= j THEN
              w:=a[i]; a[i] := a[j]
```

```
                a[j]:=w; i := i+1 ; j:=j-1
            END
        UNTIL i>j
        IF L<j THEN Sort (L,j); END;
        IF i < R THEN Sort(i,R); END
    END Sort;
```

Análisis de la clasificación rápida:

Tras seleccionar el límite x se recorre todo el arreglo, por lo que se realizan ***n*** comparaciones. El número de intercambios es más complicado, pero se puede aproximar. Con un límite fijo x, el número previsto de operaciones de intercambio es igual al de elementos en la parte izquierda de la partición, es decir, n-1 multiplicado por la probabilidad de que ese elemento haya alcanzado su lugar mediante un intercambio. Habrá tenido un intercambio si el elemento ha sido antes de la partición de la derecha; la probabilidad de ello es *(n-(x-1)) / n.* el número esperado de intercambios es, pues, el promedio de esos valores previstos sobre todos los límites posibles x.

$$M = \sum_{1}^{n} x:(x-1)*(n-(x-1))/n = \sum_{0}^{n-1} u:u(n-u)/n^2$$

$$= n*(n-1)/2n - (2n^2 - 3n + 1)/6n = (n-1/n)/6$$

Suponiendo que siempre escojamos como límite la mediana, cada proceso de partición divide en dos mitades al arreglo y el número de pases necesarios será log n. El número total resultante de comparaciones será entonces n*log n y el de intercambios n* log (n)/6.

El rendimiento promedio de la clasificación rápida es inferior al caso óptimo por un factor de apenas 2*ln(2), escogiendo al azar el límite.

El caso peor se da cuando se escoge como comparando el elemento más grande de una partición. Después de cada pase se divide un segmento de n elementos en una partición izquierda con n-1 elementos y una partición derecha con un sólo elemento. El resultado es que hay n divisiones necesarias y el rendimiento en este caso es de orden $n^{2.}$

De esto se deduce que un paso importante es la elección del comparando x. Hoare recomienda realizar de forma aleatoria la elección de x o escogerlo como la media de una pequeña muestra de, por ejemplo, tres llaves. Elección que no influye en el caso promedio pero mejora mucho el peor caso.

Clasificación por mezcla directa.

En este método, dada una secuencia *a* se realizan los procesos:

1- Dividir la secuencia *a* en dos mitades, denominadas *b* y *c*.
2- Mezclar *b* y *c* combinando cada elemento en pares ordenados.
3- Llamar *a* a la secuencia mezclada y repetir los pasos 1 y 2, combinando los pares ordenado en cuádruplos ordenados.
4- Repetir los pases anteriores, duplicando la longitud de la secuencias hasta que quede ordenada toda la secuencia.

Por ejemplo:

Dada la secuencia *a* = 44,55,12,42,94,18,06,67

En el paso 1: *b* = 44,55,12,42
c = 94, 18,06,67

La mezcla de los componentes individuales en pares ordenados da:
a = 44, 94’ 18,55’ 06, 12’ 42, 67

Dividiendo por la mitad y combinando pares ordenados se obtiene
a = 06, 12, 44, 94’ 18, 42, 56, 67

Una tercera operación de división y mezcla produce el resultado:
a = 06, 12, 18, 42, 44, 56, 67, 94

Este proceso se denomina mezcla de tres cintas. Sin embargo las fases de división no contribuyen a la clasificación, ya que no permutan elementos, por lo que son improductivas, a pesar de

que constituyen la mitad de todas las operaciones de copiado. Pueden eliminarse por completo al combinar la fase de división con la de mezcla. En vez de combinar en una secuencia, el resultado del proceso se redistribuye al instante en dos cintas, que constituyen las fuentes del pase subsecuente. En contraste con la clasificación por mezcla en tres cintas, esta técnica se conoce como mezcla de una sola fase o mezcla balanceada. Es evidentemente superior ya que se necesitan la mitad de operaciones de copiado; aunque necesita de una cuarta cinta.

Análisis de la clasificación por mezcla:

Cada paso duplica el tamaño de las subsecuencias *p* y la clasificación termina cuando $p \geq n$, por lo que incluye [log n] pases. Por definición cada pase copia el conjunto entero de n elementos exactamente una vez, por lo que el número total es:

$$M = n * \lceil \log n \rceil$$

El número de comparaciones de llaves es menor que M, ya que no intervienen comparaciones en las operaciones de copiado. Sin embargo como esta técnica de clasificación se suele aplicar cuando se utilizan medios de almacenamiento periféricos, la computación de las operaciones de movimientos domina a las comparaciones de varios órdenes de magnitud, por lo que el número de comparaciones tiene escaso interés práctico.

El algoritmo de clasificación por mezcla no es inferior a las otras técnicas avanzadas, por ejemplo es superior a la clasificación por montón pero inferior a al rápida.

Clasificación polifásica.

La ordenación de una secuencia de n corridas da como salida n corridas ordenadas, por lo que no es necesario especificar las llaves de cada corrida, basta con indicar el número de ellas que hay en la secuencia. Este método usa n cintas de salida, una de las cuales estará vacía, y n-1 contendrán el reparto de las corridas.

Comienza el proceso de mezcla que se recoge en la cinta libre hasta que una de la n-1 cintas quede vacía. El proceso se repite hasta que una de las cintas contendrá las corridas y las demás estén vacías. El proceso para 21 corridas y 3 cintas es:

f_1	f_2	f_3	Suma
13	8	0	21
5	0	8	13
0	5	3	8
3	2	0	5
1	0	2	3
0	1	1	2
1	0	0	1

El apartado suma contiene el valor de la suma de las corridas de cada nivel, que nos da la sucesión 1,2,3,5,8,13,21. Que corresponde a los números de Fibonacci, donde cada número de la serie es la suma de sus dos antecesores.

La clasificación polifásica es más eficiente que la combinación balanceada porque, con N secuencias, siempre opera con una mezcla de N-1-ple en vez de N/2-ple. A medida que el número de pases requeridos se aproxima a Log_N n, donde n es el número de elementos por clasificar y N el grado de las operaciones de combinación. La clasificación polifásica promete un notable mejoramiento respecto a la mezcla balanceada.

Pilas mediante arreglos.

Una pila se define como una estructura dinámica de datos en la que el último elemento en entrar es el primero en salir (LIFO). Los elementos están ordenador y se añaden y suprimen por un único extremo, conocido como tope o cabeza de pila. Para su implementación se utiliza como estructura de datos:

```
CONST MaxPila = 100;
TipoIndice = 1 .. MaxPila;
```

```
TYPE TipoPila = RECORD
                    Datos : ARRAY [TipoIndice] OF Tipo_Elemento
                    Cima : TipoIndice
                     END
VAR Pila: TipoPila;
```

La pila se representa con un registro con dos campos. El campo de datos dispone de un arreglo en el que se almacenan los elementos de la pila, y el campo cima que es un entero que indica la posición del tope de la pila.

Iniciar Pila

```
PROCEDURE Inicia_Pila (VAR Pila: TipoPila);
BEGIN
  Pila.Cima:=0
END Inicia_Pila;
```

Meter Pila

```
PROCEDURE Meter_Pila(VAR Pila: TipoPila; Nuevo : Tipo_Elemento);
BEGIN
  Pila.Cima := Pila.cima+1
  Pila.Datos [Pila.Cima]:= Nuevo;
END Meter_Pila;
```

Sacar Pila

```
PROCEDURE Sacar_Pila (VAR  Pila:TipoPila; VAR Dato:Tipo_Elemento);
BEGIN
 Dato := Pila.Datos[Pila.Cima];
 Pila.Cima : = Pila.Cima - 1;
END Sacar_Pila
```

Consulta de Pila Vacía

```
PROCEDURE Pila_Vacia (Pila: Tipo_Pila): BOOLEAN;
BEGIN
  RETURN Pila.Cima = 0
END Pila_Vacia;
```

Pila Llena

```
PROCEDURE Pila_Llena (Pila: Tipo_Pila): BOOLEAN;
BEGIN
  RETURN Pila.Cima = MaxPila
END Pila_Vacia;
```

Consultar Pila

```
PROCEDURE Consultar_Pila (VAR Pila: Tipo_Pila; VAR Dato: Tipo_Elemento);
BEGIN
 Dato := Pila.Datos[Pila.Cima]
END Consultar_Pila;
```

Colas mediante arreglos

Una cola se define como una estructura dinámica de datos, con los elementos ordenados y en la que el primer elemento en entrar es el primero en salir (FIFO). Los elementos se añaden por el final de la cola y se suprimen por el principio. Es necesario almacenar el principio y final de la cola, ya que si consideramos que el primer elemento de la cola está en la primera posición del arreglo, estaremos obligados a mover todos los elementos si queremos sacar el primero, lo que tiene un coste inadmisible.

El índice que indica el final de la cola será FI, el que indique el frente FR, de forma que FI indica el último elemento introducido y aumenta cuando se introduce un elementos, FR indica el primer elemento introducido y aumenta al eliminarlo. Con esta estructura los espacios del arreglo donde hay

eliminaciones quedan sin utilizar, ya que los índices nunca se decrementan. Para evitarlo se sustituye la estructura lineal del arreglo por una circular, que sin embargo produce problemas para distinguir si la cola esta llena o vacía.

Para solucionarlo se deja una posición vacía en el arreglo. De este modo de define FR de manera que apunte a un elemento que nunca se rellena, el anterior al primero de la cola, y FI tal que apunta al último elemento introducido:

Cola vacía → FR = FI

Cola Llena → FR = FI + 1 ó FI = MAXCOLA y FR = 1

De este modo la estructura de cola se define como sigue:

```
CONST MaxColan = 100;
      MaxCola = MaxColan + 1
TipoIndice = 1 .. MaxCola;
TYPE TipoCola = RECORD
                  Datos : ARRAY [TipoIndice] OF Tipo_Elemento
                  FR, FI: TipoIndice
                   END
VAR Cola: TipoCola;
```

Y los procedimientos auxiliares serían:

Iniciar Cola

```
PROCEDURE Inicia_Cola (VAR Cola: TipoCola);
BEGIN
  Cola.FI:= MaxCola;
  Cola.FR := MaxCola;
END Inicia_Pila;
```

Meter Cola

```
PROCEDURE Meter_Cola(VAR Cola: TipoCola; Nuevo : Tipo_Elemento);
BEGIN
  IF Cola.FI = MaxCola THEN Cola.FI := 1
  ELSE Cola.FI := Cola.FI +1
  END;
  Cola.Datos [Cola.FI]:= Nuevo;
END Meter_Cola;
```

Sacar Cola

```
PROCEDURE Sacar_Cola (VAR  Cola: TipoCola; VAR Dato:Tipo_Elemento);
BEGIN
 IF Cola.FR = MaxCola THEN Cola.FR := 1
  ELSE Cola.FR := Cola.FR +1
  END;
  Dato := Cola.Datos [Cola.FR];
END Sacar_Cola
```

Consulta de Cola Vacía

```
PROCEDURE Cola_Vacia (Cola: Tipo_Cola): BOOLEAN;
BEGIN
  RETURN Cola.FI =Cola.FR
END Cola_Vacia;
```

Cola Llena

```
PROCEDURE Cola_Llena (Cola: Tipo_Cola): BOOLEAN;
BEGIN
  IF Cola.FI = MaxCola THEN
    RETURN Cola.FR = 1
  ELSE
    RETURN Cola.FR = Cola.FI + 1
END Cola_Vacia;
```

Listas enlazadas.

Son estructuras de datos dinámicas que se construyen con nodos. Un nodo es un registro con dos campos, uno de ellos contiene las componentes (data) el otro es un valor que señala al siguiente nodo (next). Con variables de este tipo se puede conseguir estructuras de datos con un número de elementos que varía durante la ejecución. El campo enlace será un puntero y los nodos variables referenciadas:

```
TYPE Ptr_Nodo = POINTER TO Nodo
     Nodo = RECORD
              Next : Ptr_Nodo
              Data : Tipo_Dato
             END;
VAR Lista : Prt_Nodo;
```

Inserción por cabeza:

1- Crear un nuevo nodo.
2- Introducir el dato
3- Realizar los enlaces adecuados.

```
PROCEDURE Insertar_Cabeza (VAR Lista: Ptr_Nodo; Nuevo_dato: Tipo_dato);
VAR Nuevo_Nodo : Ptr_Nodo
BEGIN
  ALLOCATE (Nuevo_Nodo, SIZE (Nodo));
  Nuevo_Nodo^Data : = Nuevo_Dato;
  IF Lista = NIL THEN (* La lista no existe y se debe crear *)
    Lista := Nuevo_Nodo;
    Lista^.Next := NIL
  ELSE
    Nuevo_Nodo^.Next : = Lista;
    Lista : = Nuevo_Nodo
  END;
END Insertar_Cabeza.
```

Inserción al final:

Se llega al final de la lista y se realiza la inserción:

```
PROCEDURE Insertar_Final (VAR Lista: Ptr_Nodo; Nuevo_Dato: Tipo_Dato);
VAR Nuevo_Nodo, Actual : Ptr_Nodo;
BEGIN
  ALLOCATE (Nuevo_Nodo, SIZE(Nodo));
  Nuevo_Nodo^.Data:= Nuevo_Dato;
  Nuevo_Nodo^.Next := NIL;
  Actual := Lista ;
  WHILE Actual <>NIL DO
    Actual : = Actual ^.Next
  END;
  Actual^.Next:= Nuevo_Nodo
END Insertar_Final.
```

Suprimir por la cabeza:

1- Apuntar al primer elemento con un nodo auxiliar.
2- Apuntar al segundo elemento con la lista
3- Liberar de memoria el nodo a suprimir.

```
PROCEDURE Suprimir_Cabeza (VAR Lista: Ptr_Nodo; VAR Dato: Tipo_Dato);
VAR Aux: Ptr_Nodo
BEGIN
  Aux : = Lista;
```

```
  Dato : = Lista^.Data;
  Lista := Lista^.Next;
  DEALLOCATE ( Aux, SIZE (Nodo))
END Suprimir_Cabeza.
```

Implementación dinámica de pilas y colas.

PILAS

Se parte de una lista enlazada definida:

```
TYPE Tipo_Pila =POINTER TO Nodo;
        Nodo = RECORD
                Next : Tipo_Pila;
                Data : Tipo_Dato;
        END;
```

Los procedimientos auxiliares son:

```
PROCEDURE Meter_Pila (VAR Pila: Tipo_Pila; Nuevo_Dato: Tipo_Dato);
VAR Nuevo_Nodo: Ptr_Nodo;
BEGIN
  ALLOCATE (Nuevo_Nodo, SIZE(Nodo));
  Nuevo_Nodo^.Data:= Nuevo_Dato;
  Nuevo_Nodo^.Next:= Pila;
  Pila: = Nuevo_Nodo;
END Meter_Pila.

PROCEDURE Sacar_Pila (VAR Pila: Tipo_Pila; Nuevo_Dato: Tipo_Dato);
VAR Aux : Ptr_Nodo;
BEGIN
  Aux : = Pila;
  Dato : = Pila^.Data;
  Pila := Pila^.Next;
  DEALLOCATE ( Aux, SIZE (Nodo))
END Sacar_Pila.

PROCEDURE Inicia_Pila( VAR Pila: Tipo_Pila);
BEGIN
  Pila := NIL
END Inicia_Pila

PROCEDURE Pila_Vacia (Pila: Tipo_Pila): BOOLEAN;
BEGIN
  RETURN Pila = NIL
END Pila_Vacia.
```

COLAS

Si se usa una lista enlazada la inserción y supresión por el final precisa de un bucle para encontrarlo, con un número de comparaciones igual al número de elementos. Para mejorar el comportamiento se utiliza una estructura dinámica con un puntero externo al principio y final.

```
TYPE Ptr_Nodo = POINTER TO Nodo;
        Nodo = RECORD
                Next : Ptr_Nodo
                Data : Tipo_Dato
                END;
Tipo_Cola = RECORD
                Frente, Final : Ptr_Nodo
            END
```

Con esta estructura la operación *meter cola* se realiza al final de la cola y *sacar cola* al principio.

```
PROCEDURE Inicia_Cola (VAR Cola:Tipo_Cola);
BEGIN
  Cola:Final := NIL
END Inicia_Cola
PROCEDURE Meter_Cola (VAR Cola_: Tipo_Cola; Nuevo_Dato: Tipo_Dato);
VAR Nuevo_Nodo: Ptr_Nodo;
BEGIN
  ALLOCATE (Nuevo_Nodo, SIZE (Nodo));
  Nuevo_Nodo^.Data := Nuevo_Dato;
  Nuevo_Nodo^.Next : = NIL
  IF Cola.Final = NIL THEN
        Cola.Frente := Nuevo_Nodo
  ELSE Cola.Final^.Next : = Nuevo_Nodo
  END
END Meter_Cola

PROCEDURE Sacar_Cola (VAR Cola :Tipo_Cola; Dato : Tipo_Dato);
VAR Aux : Ptr_Nodo
BEGIN
  Aux : = Cola.Frente;
  Dato := Cola.Frente^.Next;
  IF Cola.Frente = NIL THEN
        Cola.Final : = NIL
  END;
  DEALLOCATE (Aux, SIZE (Nodo));
END Sacar_Cola

PROCEDURE Cola_Vacia (Cola: Tipo_Cola) : BOOLEAN;
BEGIN
  RETURN Cola.Final = NIL
END Cola_Vacia.
```

Arboles perfectamente balanceados.

Son aquellos árboles binarios en los que dado un nodo cualquiera, el número de nodos de sus subárboles izquierdo y derecho difieren como máximo en uno.

El procedimiento para construir un árbol perfectamente balanceado es:

```
PROCEDURE tree ( n : INTEGER ) : Ptr;
        VAR newmode : Ptr;
        x, nl, nr : INTEGER;
BEGIN
        IF n = 0 THEN newmode = NIL
        ELSE nl = n DIV 2 ; nr : = nl - 1;
                ReadInt ( x ); ALLOCATE ( nexmode, SIZE ( Node ));
                WITH newmode ^  DO
                        key : = x ; left : = tree ( nl ) ; rigth : = tree ( nr )
                END;
        END;
        RETURN newmode
        END tree.
```

Con la declaración de datos siguientes :

```
TYPE Ptr = POINTER TO Node;
Node = RECORD key : INTEGER;
                left, right : Ptr
```

```
    END;
VAR n : INTEGER; root : Ptr;
```

Y la primera llamada al procedimiento recursivo tree será *root : = tree(n)*; donde n será el número de nodos inicial.

Arboles balanceados.

El criterio de equilibrio propuesto por Adelson-Velski y Landis es el siguiente: Un árbol está balanceado si y sólo si en cada nodo las alturas de sus dos subárboles difieren como máximo en uno. Los árboles que satisfacen esta condición reciben el nombre de árboles AVL. En los árboles balanceados pueden efectuarse operaciones en O(log n) unidades de tiempo, aún en el peor caso:

1- Localizar un nodo con una llave determinada.
2- Insertar un nodo con una llave dada.
3- Eliminar el nodo con una llave determinada.

Esto es consecuencia directa de un teorema que garantiza que un árbol balanceado nunca tendrá una altura mayor que el 45 % respecto a su equivalente perfectamente balanceado, por muchos nodos que haya. Denotando la altura de un árbol balanceado con n nodos por $h_b(n)$ entonces:

$$\log(n+1) \leq h_b(n) < 1.4404 * \log(n+2) - 0.3828$$

Se alcanza un óptimo si el árbol está perfectamente balanceado para $n= 2^{k-1}$. Denotando T_h a un árbol de altura h . Entonces T_0 es un árbol vacío y T_1 es el árbol con un sólo nodo. Para construir un árbol T_h con h>1 se colocará en la raíz dos subárboles que tengan un número mínimo de nodos. De ahí que los subárboles también sean T, esta forma de composición se parece a la de los números de Fibonacci, por lo que se denominan árboles de Fibonacci, y se definen como sigue:

1- El árbol vacío es el árbol de Fibonacci de altura 0.
2- El nodo individual es el árbol de Fibonacci con altura 1.
3- Si T_{h-1} y T_{h-2} son árboles de Fibonacci con altura h-1 y h-2 entonces $T_h = <T_{h-1}, x, T_{h-2}>$ es un árbol de Fibonacci.
4- No hay otros árboles que sean árboles de Fibonacci.

El número de nodos de T_h se define por la siguiente relación de recurrencia:

$N_0= 0$, $N_1=1$

$N_h= N_{h-1}+1 + N_{h-2}$

Los N_i son los números de nodos en los cuales puede darse el peor caso (límite superior de h) y se les llama números de Leonardo.

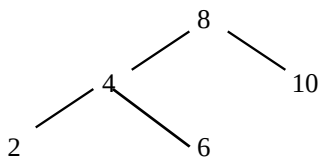
Inserción en árboles balanceados:

Cuando un nuevo nodo se inserta en un árbol balanceado pueden distinguirse tres casos con una raíz r con subárboles de izquierda y derecha L y R. Suponiendo que el nuevo nodo se inserta en L haciendo que su altura aumente en uno:

1- $h_L = h_R$: L y R tendrán una altura desigual pero cumplen el criterio de equilibrio.
2- $h_L < h_R$: L y R tendrán igual altura, el equilibrio ha mejorado.
3- $h_L > h_R$: No se cumple el criterio de equilibrio y se debe reestructurar el árbol.

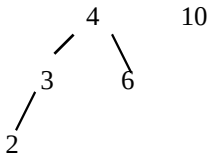
En los dos primeros casos, unas simples transformaciones reestructuran el equilibrio. Los únicos movimientos permitidos son los que ocurren en la posición vertical, mientras que las posiciones horizontales relativas de los nodos y subárboles permanecen inalteradas.

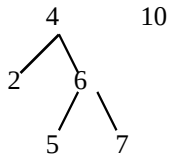
CASO ORIGINAL



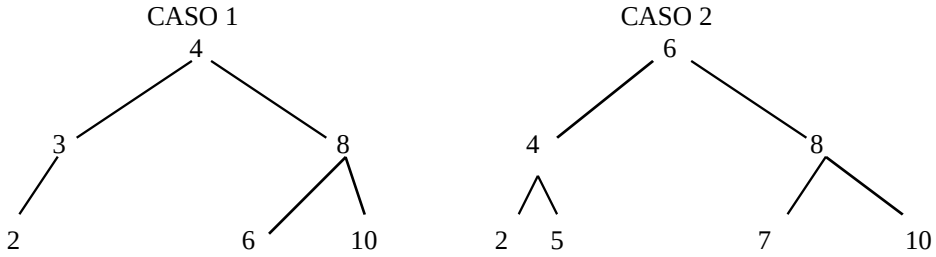
CASO 1

CASO 2





Solución :



Un algoritmo de inserción y balanceado depende de la manera en que se guarda la información referente al equilibrio del árbol. Una solución es guardar esa información en la estructura del árbol. En ese caso se deberá redescubrir un factor de balance del nodo cada vez que se vea afectado, lo que produce costes excesivamente altos. El otro extremo consiste en atribuir a cada nodo un factor de balance explícitamente almacenado. La definición de nodo queda de forma:

```
TYPE Ptr = POINTER TO Node ;
TYPE Balance = [ -1 .. +1 ]
TYPE Node = RECORD key : INTEGER;
                   count : INTEGER ;
                   left, right : Ptr ;
                   bal : Balance
                 END
```

El proceso de inserción consta de tres partes consecutivas :

1- Seguir la trayectoria de búsqueda hasta verificar que la llave no existe en el árbol.
2- Insertar el nodo y determinar el factor de equilibrio resultante.
3- Retroceder a lo largo de la trayectoria de búsqueda y verificar el factor de equilibrio en cada nodo. Si es necesario se hace rebalanceo.

El procedimiento describe la operación de búsqueda que se requiere en cada nodo, como es de naturaleza recursiva fácilmente admite una operación cuando retorna de la trayectoria de búsqueda. En cada caso, hay que transmitir información sobre si ha aumentado la altura del subárbol. Por tanto, se aplica la lista de parámetros del procedimiento mediante una **h** booleana, lo que significa que ha aumentado la altura del subárbol.

Suponiendo que el proceso está retornando a un nodo p^a partir de la rama izquierda, con la indicación de que ha aumentado su altura. A continuación se debe de distinguir entre las tres condiciones relativas al subárbol antes de la inserción:

1. $h_L<h_R$ p^bal = +1, el desequilibrio anterior en p ha sido equilibrado con la inserción.
2. $h_L=h_R$ p^bal = 0, el peso de inclina hacia la izquierda
3. $h_L>h_R$ p^bal = -1 , se necesita rebalanceo

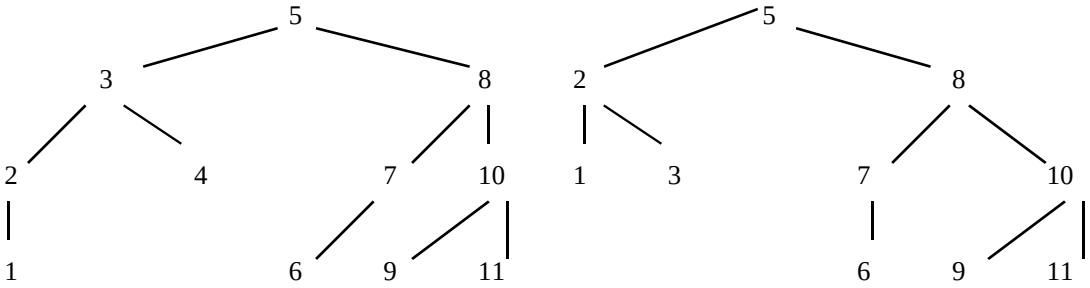
En el tercer caso si ese nodo tiene además un subárbol de la izquierda más alto que el de la derecha, tendremos un caso como el primero del ejemplo; de lo contrario el caso será del segundo tipo. Las operaciones de rebalanceo se expresan como secuencias de las reasignaciones de punteros. Estos se intercambian cíclicamente y dan como resultado una rotación de punteros, además hay que actualizar los respectivos factores de equilibrio de nodos.

La complejidad de las operaciones de balanceo indica que los árboles balanceados deben utilizarse sólo si las recuperaciones de información son mas frecuentes que las eliminaciones. Esto es debido a que los nodos suelen instrumentarse como registros densamente compactados a fin de ahorrar espacio. La velocidad de acceso y de actualización de los factores de balanceo suelen ser un factor decisivo en la eficiencia de la operación de rebalanceo.

Eliminación en árboles balanceados.

Los casos más sencillos son los nodos terminales y los que tienen un sólo descendiente. Si el nodo que debe suprimirse tiene dos subárboles, se reemplaza por el nodo del extremo derecho de su subárbol de la izquierda. Como en la inserción se agrega un parámetro de variable booleana h, que indica si la altura del árbol ha sido reducida. El rebalanceo se considerar sólo cuando h es verdadera después de encontrar y suprimir un nodo, o si se disminuye la altura de un subárbol.

Por ejemplo:



La eliminación de un elemento en un árbol balanceado se realiza con un coste O(log n). La diferencia fundamental entre los procesos de inserción y eliminación es que mientras la inserción de una llave puede producir a lo sumo una rotación (de dos o tres nodos), la eliminación puede requerir una rotación de cada nodo a lo largo de la trayectoria de búsqueda.

Arboles de búsqueda óptimos.

Hay casos en que la información sobre las probabilidad de acceso a llaves individuales está disponible. Estos casos tienen la característica de que las llaves siempre permanecen inalteradas, es decir, que el árbol de búsqueda no está sujeto a inserciones ni eliminaciones, sino que conserva una estructura constante. En estos árboles los nodos a los que se accede frecuentemente se denominan nodos pesados; los menos visitados son nodos ligeros.

Para organizar el árbol de forma que el número de accesos sea mínimo se debe modificar la definición de trayectoria, dándole un determinado peso a cada nodo y suponiendo que la raíz se halla en el nivel 1 en vez del 0 ya que representa la primera comparación a lo largo de la trayectoria de búsqueda. La longitud de trayectoria pesada (interna) es la suma de todas las que van de la raíz a cada nodo, ponderadas por la probabilidad de acceso del nodo:

$$P = \sum i{:}1 \le i \le n{:}\, p_i * h_i$$

h_i es el nivel del nodo i. La meta consiste en minimizar la longitud de trayectoria pesada para determinar la distribución de probabilidad. Si también se conoce la probabilidad q_i de un argumento de búsqueda x, situado entre las dos llaves k_i y k_{i+1}, esta información puede modificar la estructura del árbol de búsqueda optimo. Por ello para generalizar el problema se deben de tener en cuenta las búsquedas infructuosas. De esta forma la búsqueda promedio global de la trayectoria pesada es:

$$P = \sum i{:}1 \le i \le n{:}\, p_i * h_i + \sum j{:}0 \le j \le m{:}\, q_j * h'_i$$

donde

$$\sum i{:}1 \le i \le n{:}\, p_i + \sum j{:}0 \le j \le m{:}\, q_j = 1$$

h_i es el nivel del nodo (interno) i, h'_j es el nivel del nodo externo j. La longitud promedio de trayectoria ponderada se llama costo del árbol de búsqueda. El árbol de búsqueda que requiere menos costo entre todos los árboles con un determinado conjunto de llaves k_i y probabilidades p_i y q_i recibe el nombre de árbol óptimo. Para encontrar un árbol óptimo no se necesita que las p y las q sumen 1. Estas probabilidades se determinan mediante experimentos, y en lugar de ellas se utilizan los conteos de frecuencia denotados:

a_i = número de veces que el argumento de búsqueda x es igual a k_i

b_j = número de veces que el argumento de búsqueda x se encuentra entre k_j y k_{j+1}.

Utilizando estos conteos se usa P para denotar la *longitud acumulada de trayectoria pesada* en vez de la longitud promedio de trayectoria:

$$P = \sum_{i=1}^{n} a_i \times h_i + \sum_{j=0}^{m} b_j \times h'_j$$

Como el número de configuraciones de n nodos crece exponencialmente con n, parece imposible encontrar el óptimo para una n grande. Sin embargo los árboles óptimos tienen la propiedad de que todos sus subárboles también lo son, lo que facilita la labor. Si se comienza por nodos individuales, se pueden encontrar árboles cada vez más grandes.

La llave del algoritmo de construcción de árboles óptimos es la ecuación:

$P = P_L + W + P_R$

$W = (\sum i : 1 \le i \le n : a_i) + (\sum j : 0 \le j \le m : b_j)$

P es la longitud de trayectoria pesada (ponderada) de un árbol y P_L y P_R las de los subárboles de la izquierda y derecha de su raíz. W es el número total de intentos de búsqueda, se llama peso del árbol. Su longitud promedio de trayectoria es P/W.

Hashing

La organización de datos usada en esta técnica es la estructura de arreglo. H es un mapeo que transforma las llaves en índices de arreglo, y por eso se emplea la designación transformación de llaves. No es necesario basarse en ningún procedimiento de asignación dinámica.

La dificultad al utilizar una transformación de llaves es que el conjunto de posibles valores básicos resulta mucho mayor que el conjunto de direcciones disponibles de memoria (índices del arreglo). Por ejemplo si deseamos almacenar nombres de hasta 16 letras como llaves que identifican a las personas en un grupo de mil. Habrá 28^{16} llaves posibles que deben de mapearse en 10^3 índices posibles. H es una función de muchos contra uno. Si tenemos una llave k, el primer paso en una operación de búsqueda es calcular su índice asociado h =H(k); el segundo verificar si el elemento con la llave k es identificado por h en el arreglo T, o sea, verificar si T [H (k)] . llave = k.

Elección de una función de transformación de llaves.

Un requisito básico de una función de transformación es que distribuya las llaves lo más uniformemente posible sobre los valores del índice. La distribución no está ligada a patrón alguno y es conveniente que dé la impresión de ser aleatoria. Esta propiedad da al método el nombre de Hashing (picadillo). H recibe el nombre de función de transformación y debe de ser calculable de modo eficiente.

Por ejemplo la función H (k) = ORD (k) MOD N, siendo N el número de direcciones posibles de almacenamiento, tiene la propiedad de que los valores de las llaves estarán distribuidos uniformemente sobre el índice. Si N fuese potencia de 2 las palabras que difieren sólo en unos pocos caracteres se mapearán seguramente en índices iguales, con una distribución muy poco uniforme, por lo que es recomendable que N sea un número primo.

Otra función de transformación consiste en aplicar operaciones lógicas como la o-exclusiva para algunas partes de las llaves representadas como una secuencia de dígitos binarios (poco utilizada).

Manejo de las colisiones.

Si resulta que un elemento de una tabla correspondiente a determinada llave no es el elemento deseado, existe colisión, es decir hay dos elementos que mapean en el mismo índice. Entonces es necesario un segundo sondeo. Hay varios métodos para generar los índices secundarios. Uno es ligar todos los elementos con el índice primario idéntico H (k) en una lista ligada. A esto se le llama encadenamiento directo. Los elementos de la lista pueden estar en la tabla primaria o no; en el segundo caso, se da el nombre de área de desbordamiento al almacenamiento donde están asignados. Este método tiene la desventaja de que deben conservarse las listas secundarias y de que cada elemento ha de reservar espacio para un puntero a su lista de elementos en colisión.

Otra solución del problema de las colisiones estriba en prescindir de las ligas y en su lugar limitarse a observar las otras entradas en la misma tabla hasta encontrar una posición abierta entonces se supone que la llave no está en la tabla. Este método recibe el nombre de lista abierta. La secuencia de los índices de las exploraciones secundarias siempre debe de ser igual para una llave determinada. El esquema del algoritmo en una búsqueda en tablas es:

```
h : = H ( k ) ; i : = 0 ;
REPEAT
        IF T [ h ].key = k THEN  elemento encontrado
        ELSIF T [h].key = free THEN el elemento no está en la tabla
        ELSE (* colisión *)
```

i : = i + 1 ; h : = H (k) + G (i)
END
UNTIL ***encontrado o no está en la tabla, o tabla llena***

El método más sencillo entre las funciones para solucionar las colisiones consiste en ensayar la siguiente localización (suponiendo que la tabla es circular) hasta encontrar el elemento con la llave especificada o una localización vacía. Por tanto G (i); los índices H_i usados para hacer la exploración son :

h_0= H (k)
h_1= (h_0 + i) MOD N, i=1 … N-1

Se llama exploración lineal y su desventaja es que las entradas se agrupan alrededor de las llaves primarias. Debe de elegirse una función G que otra vez distribuya uniformemente las llaves en el resto del conjunto de localizaciones. En la práctica suele ser demasiado costoso. Otra solución consiste en aplicar una función cuadrática tal que la secuencia de índices de la búsqueda sea :

h_0= H (k)
h_1= (h_0 + i^2) MOD N, i >0

Este método se llama exploración cuadrática, y evita la agrupación primaria aunque no se necesiten cálculos adicionales. Su desventaja es que no realiza la búsqueda en todas las tablas, es decir, después de la inserción puede que no se encuentre una posición libre aunque realmente exista. Prácticamente este inconveniente carece de importancia pues tener que realizar N / 2 exploraciones secundarias y evasiones de colisión es muy raro y sólo ocurre si la tabla está casi llena.

Aunque el método de transformación de llaves es más eficiente que los árboles, presenta una desventaja. Luego de analizar las palabras si las queremos presentar por orden alfabético, debemos de realizar la ordenación antes de la presentación. En consecuencia el rendimiento superior de este método, considerando sólo el proceso de recuperación, está contrarrestado en parte por las operaciones adicionales que se requieren para terminar la tarea de generar un índice ordenado.

Análisis de la transformación de llaves.

La inserción y recuperación por transformación de llaves tiene un rendimiento pésimo en el peor caso, ya que es posible que un argumento de búsqueda sea tal que las búsquedas encuentren precisamente todas las localizaciones ocupadas, sin hallar las que se desean (o libres). En promedio el número de búsquedas es pequeño y depende exclusivamente del factor de carga del array de almacenamiento.

Una llave debe ser insertada en una tabla de tamaño n que ya contiene k elementos. Suponiendo que todas la llaves tienen las misma probabilidad y que la función de transformación H las distribuye uniformemente sobre el intervalo de los índices de la tabla. La probabilidad de encontrar una localización vacía la primera vez será de (n-k)/n. Además es la probabilidad de que se necesite una sola comparación. La probabilidad de que se requiera exactamente una segunda exploración es igual a la probabilidad de una colisión en el primer intento, multiplicada por la probabilidad de hallar una localización libre la siguiente vez. En general sería:

P_1= (n - k)/n
P_2= (k/n) × (n - k)/(n-1)
P_3 =(k/n) × (n - k)/(n-1) × (n - k) × (n-2)
… … …
P_i = (k/n) * (n -k)/(n-1)×(k-2)/(n-2)× …× (n - k)/ (n-i+1)

El número esperado de exploraciones, E, requeridas después de insertar la llave k+1-iésima será entonces:

$$E_{k+1}= \sum_{i=1}^{k+1} i \times p = (n+1)/(n-k+1)$$

Puesto que el número de búsquedas requeridas para insertar un elemento es idéntico al número de las que se necesitan para recuperarlo, el número promedio E de exploraciones necesarias para acceder a una llave aleatoria de una tabla de tamaño n con m laves será:

$$E = (\sum_{k=1}^{m} E_k) / m = (n+1) \times (H_{n+1} - H_{n-m+1}) / m$$

donde H es la función armónica. H puede ser aproximada como H_n= ln(n)+g, donde g es la constante de Euler. Sustituyendo a= m/(n+1) obtenemos:

E = -ln(1-a)/a

a es aproximadamente el cociente de las localizaciones ocupadas y disponibles, llamado factor de carga, a=0 supone una tabla vacía y a=n/(n+1) una tabla llena. El número de previsto E de exploraciones para recuperar o insertar una llave elegida de modo aleatorio en función del factor de carga es:

Factor de carga	Exploración cuadrática	Exploración lineal
0.1	1.05	1.06
0.25	1.15	1.17
0.5	1.39	1.50
0.75	1.85	2.50
0.9	2.56	5.50
0.95	3.15	10.50
0.99	4.66	------

Se necesitan en promedio menos de 3 exploraciones para encontrar una llave o una localización vacía en un array lleno al 90%, factor de carga 0.9, mediante exploración cuadrática.

Inconvenientes del Hashing.

El tamaño del array es fijo y no puede ajustarse a la exigencia del momento.
La eliminación de un elemento insertado es muy difícil.

El Hashing es adecuado cuando:

1- Se conoce a priori el volumen de datos a tratar.
2- El volumen de datos experimenta pocas variaciones.
3- No debe de modificarse/ borrarse los elementos insertados

En otros caso es más optimo la utilización de organizaciones de árbol.