CONTENIDO

4.1 INTRODUCCION

4.2 ORDENACION

4.2.1 INSERTION SORT

4.2.2 SHELL SORT

4.2.3 QUICKSORT

4.2.4 COMPARACIONES

4.3 DICCIONARIOS

4.3.1 TABLAS DISPERCIÓN

4.3.2 ARBOLES DE BUSQUEDA BINARIA

4.3.3 ARBOLES RED-BLACK

4.3.4 SKIP LISTS

4.3.5 COMPARACIONES

4.4 ARCHIVOS MUY LARGOS

4.4.1 ORDENACION EXTERNA

4.4.2 B-TREES

4.1. Introducción

Los arreglos y las listas enlazadas son las dos estructuras básicas de datos más utilizadas para almacenar información. Nosotros podemos desear buscar, insertar o borrar registros en una base de datos en base a un valor clave. Esta sección examina el desempeño de estas operaciones sobre arreglos y listas enlazadas.

Arreglos (Arrays)

La Figura 4.1.1 muestra un arreglo, de siete elementos de longitud, conteniendo valores numéricos. Para buscar en el arreglo secuencialmente, podemos usar el algoritmo de la Figura 4.1.2. El número máximo de comparaciones es 7, y ocurre cuando la clave que estamos buscando está en A[6]. Si los datos se ordenan, podría hacerse una búsqueda binaria (Figura 4.1.3). Las variables Lb y Ub guardan los límites superior e inferior del arreglo, respectivamente. Nosotros comenzamos por examinar el elemento medio del arreglo. Si la clave que nosotros estamos buscando es menor que el elemento medio, entonces debe estar ubicada en la mitad inicial del arreglo. Así, nosotros colocamos Ub a (M-1). Esto restringe nuestra próxima iteración mediante el bucle a la mitad inicial del arreglo. De esta manera, cada iteración reduce a la mitad el tamaño del arreglo para ser buscado. Por ejemplo, la primera iteración dejará 3 artículos para probar. Después de la segunda iteración, habrá 1 artículo para probar. Así toma tres únicas iteraciones para encontrar cualquier número.

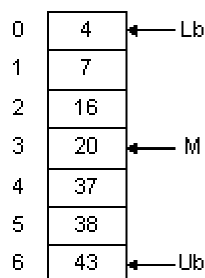


Figura 4.1.1: Un Arreglo (Array)

Este es un método poderoso. Por ejemplo, si el tamaño de un arreglo es 1023, podemos estrechar la búsqueda a 511 artículos en una comparación. Otra comparación y miraremos tan solo 255 elementos. De hecho, solo 10 comparaciones se necesitan para buscar en un arreglo que contenga 1023 elementos.

Además de buscar, nosotros podemos desear insertar o borrar entradas. Desgraciadamente, un arreglo no es lo bastante bueno para estas operaciones. Por ejemplo, para insertar el número 18 en la Figura 1.1, necesitaríamos cambiar A[3]…A[6] hacia abajo por una posición. Entonces podríamos copiar el número 18 en A[3]. Un problema similar proviene cuando se borran números. Para mejorar la eficacia de las operaciones de inserción y borrado, se pueden utilizar listas enlazadas.

```
Int function BusquedaSecuencial (Array A, int Lb, int Ub, int Key);
 begin
 for i = Lb to Ub do
   if A(i) =  Key then
     return i;
   return –1;
 end;
```

Figura 4.1.2: La Búsqueda Secuencial

```
int function BúsqeudaBinaria (Array A, int Lb, int Ub, int Key);
 begin
 do forever
  M = (Lb + Ub)/2;
  if (Key < A[M]) then
   Ub = M – 1;
  else if (Key  A[M]) then
   Lb = M + 1;
  else
   return M;
  if (Lb  Ub) then
   return –1;
 end;
```

Figura 4.1.3: La Búsqueda Binaria

Listas Enlazadas (Linked Lists)

En la Figura 1.4 hemos almacenado los mismos valores en una lista enlazada. Supongamos los punteros X y P, como muestra la figura, el valor 18 puede insertarse como se indica a continuación:

```
X->Next = P->Next;
P->Next = X;
```

La inserción (y eliminación) son más eficientes utilizando listas enlazadas. Usted puede preguntarse como es que P es configurada en el primer lugar. Bien, tuvimos que buscar la lista de una manera secuencial para encontrar la inserción señalada por X. Así, mientras mejoramos el desempeño de nuestra inserción y eliminación, ha sido sacrificando tiempo de búsqueda.

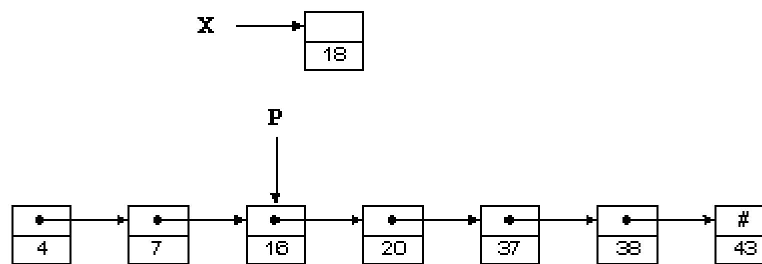


Figura 4.1.4: Una Lista Enlazada (Linked List)

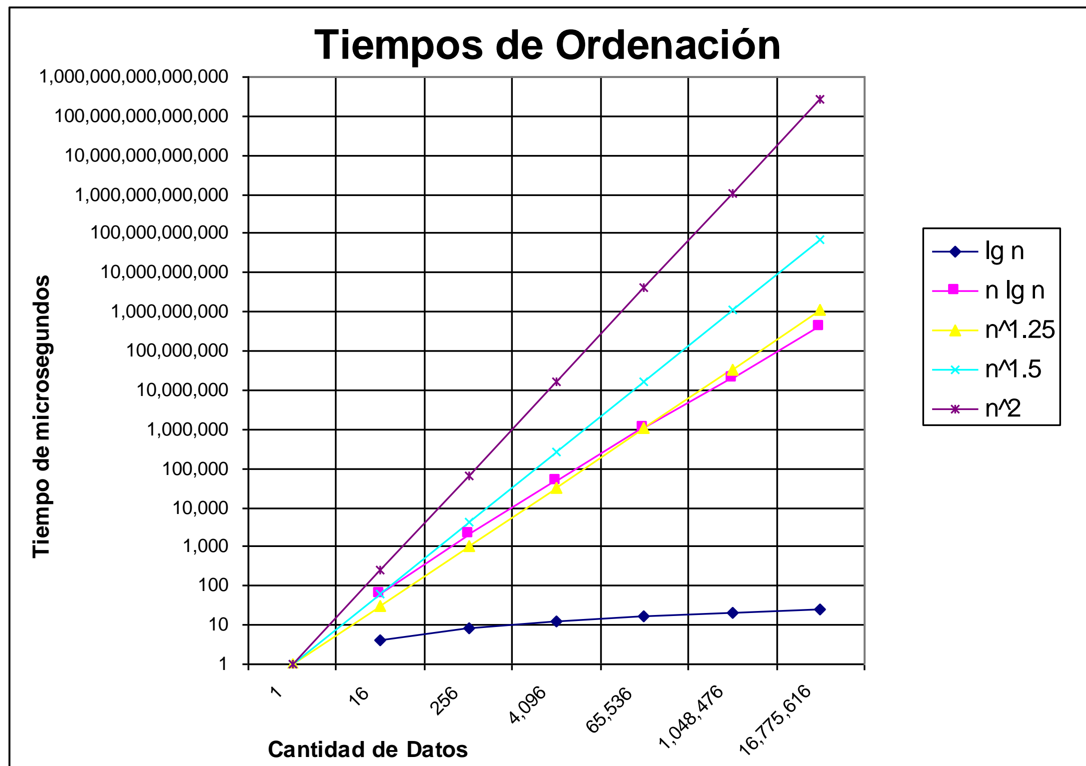
Tiempo Estimado

Varios métodos pueden utilizarse para comparar el desempeño de algoritmos. Una manera es simplemente correr varias pruebas para cada algoritmo y comparar los tiempos. Otra manera es estimar el tiempo requerido. Por ejemplo, podemos afirmar que el tiempo de búsqueda es el O(n) (Gran-Oh de n / Big-Oh of n). Esto significa que, para una n grande, el tiempo de búsqueda es no mayor del número de artículos n en la lista. La notación Gran-O no describe el tiempo exacto de participaciones del algoritmo, pero únicamente indica un límite superior sobre el tiempo de ejecución dentro de un factor constante. Si un algoritmo toma $O(n^2)$ veces, entonces el tiempo de ejecución no crece pero que el cuadrado del tamaño de la lista. Para ver el efecto que esto tiene, la Tabla 4.1.1 ilustra el tiempo de ordenación para diversas funciones. Un valor de crecimiento de O(lg n) ocurre para algoritmos parecidos a la búsqueda binaria. La función lg (logaritmo, base 2) aumenta por uno cuando n se duplica. Recuerde que podemos buscar un par de veces muchos más artículos que con una comparación en búsqueda binaria. Así, la búsqueda binaria es un algoritmo O(lg n).

n	lg n	n lg n	n^1.25	n^2
1	0	0	1	1
16	4	64	32	256
256	8	2,048	1,024	65,536
4,096	12	49,152	32,768	16,777,216
65,536	16	1,048,565	1,408,476	4,294,967,296
1,048,476	20	20,969,520	33,554,432	109,930,192,256
16,775,616	24	402,614,784	1,073,613,825	281,421,292,179,456

Tabla 4.1.1: Tiempos de Ordenación

Gráfico 4.1.1: Tiempos de Ordenación



Si los valores en la Tabla 4.1.1 representan microsegundos, entonces un algoritmo O(lg n) puede tomar 20 microsegundos para procesar 1,048,476 artículos, un algoritmo $O(n^{1.25})$ podría tomar 33 segundos, y un algoritmo $O(n^2)$ podría tomar 12 días!. En los capítulos siguientes una estimación de tiempo para cada algoritmo, usando la notación Gran-O, será incluida. Para una derivación más formal de estas fórmulas puede encontrarlas consultando la bibliografía al final.

Resumen

Como hemos visto, ordenando los arreglos pueden buscarse eficientemente usando una búsqueda binaria. Sin embargo, nosotros debemos tener un arreglo ordenado para comenzar con esto. En la próxima sección las diversas maneras para ordenar u ordenar arreglos serán examinadas. Resulta ser que esto es computacionalmente caro, y

se ha hecho una investigación considerable hacer algoritmos de ordenación tan eficientes como sea posible.

Las listas enlazadas mejoraron la eficacia de operaciones de inserción y eliminación pero las búsquedas eran secuenciales y consumían tiempo. Existen algoritmos que hacen todas estas operaciones eficientemente, y ellos serán discutidos en la sección sobre diccionarios.

4.2. Ordenación

Algunos Algoritmos serán tratados, incluyendo *insertion sort, shell sort* y *quicksort*. La ordenación por inserción (*Insertion sort*), es el método más simple, y no requiere de almacenamiento adicional. *Shell sort* es una simple modificación que aumenta significativamente su rendimiento. Probablemente el método más eficiente y popular es el *Quicksort*, y es el ideal para arreglos muy largos.

4.2.1 Ordenación por inserción (Insertion Sort)

Uno de los métodos más simples de ordenar arreglos es utilizando la *Insertion Sort*. Un ejemplo de este método ocurre siempre en nuestra vida cotidiana cuando jugamos cartas. Para ordenar las cartas en su mano, extrae una de ellas, cambia las cartas restantes, y entonces inserta la carta extraída en el lugar correcto. Este proceso se repite hasta que todas las cartas estén en la sucesión correcta. Ambos en el peor caso tienen un promedio de tiempo de $O(n^2)$. Para lecturas complementarias consulte Knuth.

Teoría

En la Figura 4.2.1(a) extraemos el 3. Entonces los elementos superiores se cambian hacia abajo hasta que encontremos el lugar correcto para insertar el 3. Este proceso se repite en la Figura 4.2.1 (b) para el número 1. Finalmente, en la Figura 4.2.1 (c), completamos la ordenación para insertar el 2 en el lugar correcto. Suponiendo hay n elementos en el arreglo, debemos indexarlo mediante *n-1* entradas. Para cada entrada, podemos necesitar examinar y cambiar hasta otras *n-1* entradas. Por esto, la ordenación es un proceso que consume mucho tiempo.

La ordenación por inserción es una ordenación *in-place*. Que es, ordenamos el arreglo *in-place.* No se requiere ninguna memoria adicional. La ordenación por inserción es también una ordenación estable. Las ordenaciones estables retienen la ordenación original de claves cuando las claves idénticas están presentes en los datos de entrada.

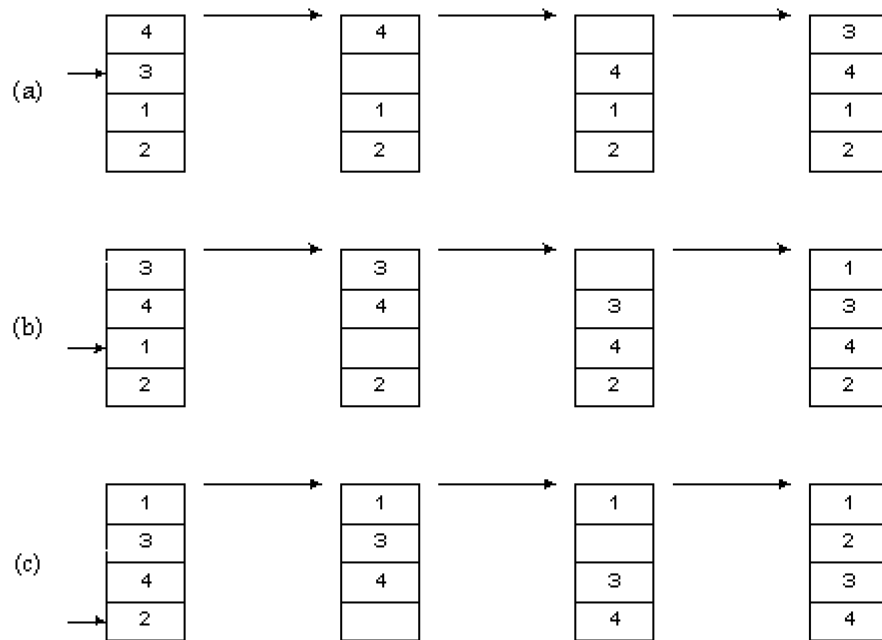


Figura 4.2.1: Insertion Sort

Implementación

Typedef ***T*** y el operador de comparación ***compGT***, deberían alternarse para reflejar los datos almacenados en la tabla. La aritmética de punteros se utilizó mejor que la referencia de arreglos, para obtener mayor eficacia.

```
#include <stdio.h>
#include <stdlib.h>

typedef int T;          /* tipo de item a ser ordenado */
typedef int tblIndex;   / tipo de subscript */
#define compGT(a,b) (a > b)

void InsertaOrdena(T *a, tblIndex lb, tblIndex ub) {
    T t;
    tblIndex i, j;

   /*********  ordena array a[lb..ub]  *********/
    for (i = lb + 1; i <= ub; i++) {
        t = a[i];

        /* intercambia elementos hasta */
        /* encontrar punto de inserción. */
        for (j = i-1; j >= lb && compGT(a[j], t); j--)
            a[j+1] = a[j];
        /* inserta */
        a[j+1] = t;
    }
}

void llena(T *a, tblIndex lb, tblIndex ub) {
    tblIndex i;
    srand(1);
    for (i = lb; i <= ub; i++) a[i] = rand();
}

int main(int argc, char *argv[]) {
    tblIndex maxnum, lb, ub;
    T *a;

   /* linea de comandos: ins maxnum; ins 2000; ordena 2000 registros */

    maxnum = atoi(argv[1]);
    lb = 0; ub = maxnum - 1;
    if ((a = malloc(maxnum * sizeof(T))) == 0) {
        fprintf (stderr, "memoria insuficiente \n");
        exit(1);
    }
    llena(a, lb, ub);
    InsertaOrdena(a, lb, ub);
    return 0;
}
```

4.2.2 Ordenación Shell (Shell Sort)

Shell sort, desarrollada por Donald L. Shell, es una ordenación *non-stable in-place*. Shell sort mejora sobre la eficiencia de la ordenación por inserción por medio de cambios rápidos a su destino. El tiempo promedio de ordenación es de $O(n^{1.25})$, mientras que en el peor de los casos tiene un tiempo de $O(n^{1.5})$. para lectura complementaria lea Knuth.

Teoría

En la Figura 4.2.2 (a) tenemos un ejemplo de ordenación por inserción. Primero extraemos el 1, cambiamos 3 y 5 hacia abajo una posición, y entonces insertamos el 1. Así, se requirieron dos cambios. En el próximo cuadro, se requieren dos cambios antes que podamos insertar el 2. El proceso continúa hasta el último cuadro, donde un total de 2+2+1=5 cambios se deben haber realizado.

En la Figura 4.2.2 (b) se ilustra un ejemplo de la Shell Sort. Comenzamos por hacer una ordenación de inserción (Insertion sort) que usa un espaciamiento de dos. En el cuadro primero examinamos los números 3 y 1. Extraemos el 1, cambiamos el 3 hacia abajo una posición, para un total de cambios igual a 1. Luego examinamos los números 5 y 2, extraemos el 2, y cambiamos el 5 hacia abajo, y entonces insertamos el 2. Después de ordenar con un espaciamiento de dos, se realiza una pasada final con un espaciamiento de uno. Esta es simplemente la ordenación tradicional por inserción. El conteo final de cambios que usa la Shell Sort es de 1+1+1=3. Por usar un espaciamiento inicial más grande que uno, éramos capaces de rápidamente cambiar valores a su destino apropiado.

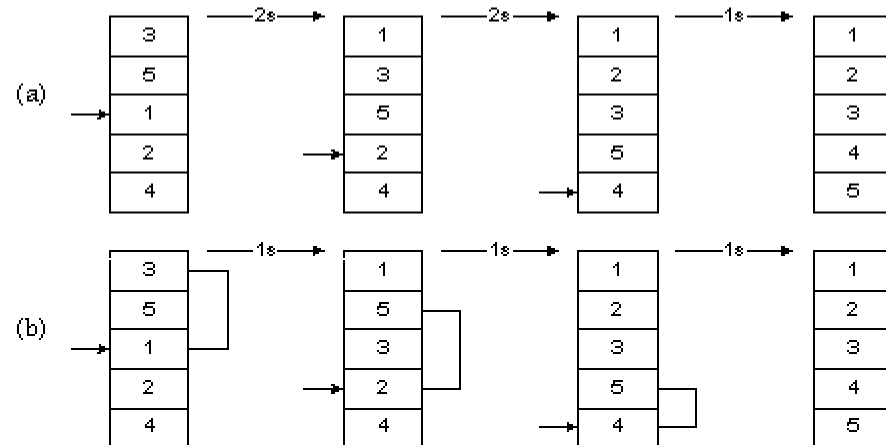


Figura 4.2.2: Shell Sort

Para implantar la Shell Sort, pueden usarse diversos espaciamientos. Típicamente el arreglo se ordena con un espaciamiento grande, el espaciamiento es reducido y el arreglo se ordena nuevamente. Sobre la ordenación final, el espaciamiento es uno. Aunque la Shell Sort sea fácil de comprender, el análisis formal es difícil. En particular, valores óptimos de espaciamiento elude las teorías. Knuth ha experimentado con varios valores y recomienda que espaciando (h) para un arreglo de tamaño N debe ser con base en la fórmula siguiente:

Dejar $h_1 = 1$, $h_s+1 = 3h_s + 1$, y pare con h_t cuando $h_{t+2} >= N$.

Así, los valores de h se computan como se indica a continuación:

$h_1 = 1$

$h_2 = (3 \text{ x } 1) + 1 = 4$

$h_3 = (3 \text{ x } 4) + 1 = 13$

$h_4 = (3 \text{ x } 13) + 1 = 40$

$h_5 = (3 \text{ x } 40) + 1 = 121$

Para ordenar 100 artículos, primero se encuentra unas h_s tales que $h_s > = 100$. Para 100 artículos, h_5 es seleccionada. Nuestro valor final (ht) es dos pasos mas abajo, o

h_3. Así, nuestra sucesión de valores de h serán 13-4-1. Una vez el valor inicial de h se ha determinado, los valores subsiguientes pueden calcularse usando la fórmula

$$h_{s-1} = h_s / 3$$

Implementación

Typedef ***T*** y el operador de comparación ***compGT*** deberían alterarse para reflejar los datos almacenados en el arreglo. Cuando calcule h, debe hacerse con cuidado para evitar underflows o rebosaduras. La porción central del algoritmo es una ordenación por inserción con un espaciamiento de **h**.

```
#include <stdio.h>
#include <stdlib.h>

typedef int T;          /* tipo de item a ser ordenado */
typedef int tblIndex;   /* tipo de subscript */

#define compGT(a,b) (a > b)

void OrdenacionShell(T *a, tblIndex lb, tblIndex ub) {
    tblIndex n, h, i, j;
    T t;

   /**************************
      ordena array a[lb..ub]  *
    **************************/

    /* calcula el incremento largo */
    n = ub - lb + 1;
    h = 1;
    if (n < 14)
        h = 1;
    else if (sizeof(tblIndex) == 2 && n > 29524)
        h = 3280;
    else {
        while (h < n) h = 3*h + 1;
        h /= 3;
        h /= 3;
    }
    while (h > 0) {
        /* ordena por inserción en incrementosde h */
        for (i = lb + h; i <= ub; i++) {
            t = a[i];
            for (j = i-h; j >= lb && compGT(a[j], t); j -= h)
                a[j+h] = a[j];
            a[j+h] = t;
        }
        /* calcula el siguiente incremento */
        h /= 3;
    }
}

void llena(T *a, tblIndex lb, tblIndex ub) {
    tblIndex i;
    srand(1);
    for (i = lb; i <= ub; i++) a[i] = rand();
```

```
}

int main(int argc, char *argv[]) {
    tblIndex maxnum, lb, ub;
    T *a;

    /* linea de comandos: shl maxnum

      shl 2000
      ordena 2000 registros
     */
    maxnum = atoi(argv[1]);
    lb = 0; ub = maxnum - 1;
    if ((a = malloc(maxnum * sizeof(T))) == 0) {
        fprintf (stderr, "memoria insuficiente \n");
        exit(1);
    }
    llena(a, lb, ub);
    OrdenacionShell(a, lb, ub);
    return 0;
}
```

4.2.3 Ordenación Rápida (Quicksort)

Aunque el algoritmo Shell Sort es significativamente mejor que le Insertion Sort, todavía queda opción de mejora. Uno de los algoritmos de ordenación más populares es el Quicksort. Quicksort se ejecuta en un tiempo de O(n lg n), y O(n^2) en el peor de los casos. Sin embargo, con las apropiadas precauciones, el peor de los casos podría tener comportamientos inesperados. El Quicksort es una ordenación *non-stable*. No requiere un espacio de almacenamiento. Para mayor referencia consulte Cormen [1990].

Teoría

El algoritmo Quicksort trabaja por particionamiento del arreglo a ser ordenado, entonces recursivamente ordena cada partición. En la Partición (Figura 4.2.3), uno de los elementos del arreglo es seleccionado como valor pivote. Los valores menores que el pivote se colocan a la izquierda de este, mientras que los mayores se colocarán a la derecha.

```
Int function Particion(Array A, int Lb, int Ub);
begin
select a pivot from A[Lb]…A[Ub];
reorder A[Lb]…A[Ub] such that:
        all valores a la izquierda del pivot son <= pivot
        all valores a la derecha del pivot son >= pivot
return pivot position;
end;

procedure QuickSort(Array A, int Lb, int Ub);
begin
if Lb < Ub then
        M = Particion(A,Lb,Ub);
        QuickSort(A,Lb,M-1);
        QuickSort(A,M+1,Ub);
End;
```

Figura 4.2.3: Algoritmo Quicksort

En la Figura 4.3.4(a), el pivot seleccionado es 3. Los índices son movidos al comienzo y al final del arreglo. El índice i comienza en la izquierda y selecciona un elemento que es mayor que el pivot, mientras el índice j comienza en la derecha y selecciona un elemento que es menor que el pivot. Estos elementos son entonces intercambiados, como muestra la Figura 4.3.4(b). Quicksort recursivamente ordena los dos subarreglos, resultando un arreglo como se muestra en la Figura 4.3.4(c).

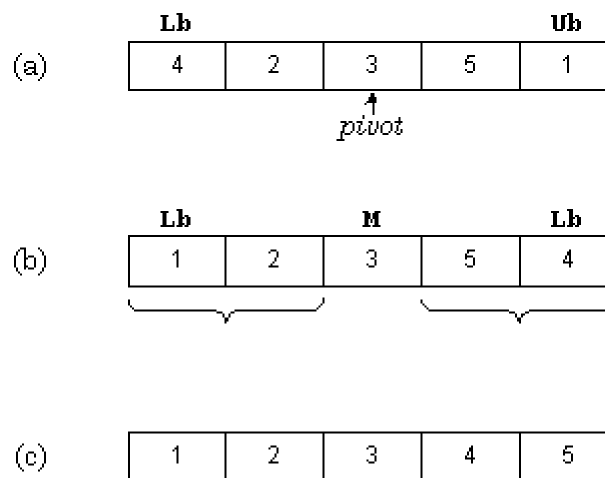


Figura 4.2.4: Ejemplo Quicksort

Como el proceso funciona, es necesario mover el pivot para que la ordenación sea correcta. En esta manera, Quicksort culmina la ordenación del arreglo. Si somos afortunados el pivot seleccionado podría ser la mediana de todos los elementos, el cual estaría justamente en la mitad del arreglo. Por un momento, asumamos que este es el caso. Entonces el arreglo es dividido en cada paso, y Partition examinaría eventualmente todos los *n* elementos, su corrida tendría un tiempo de O(n lg n).

Para encontrar el pivot, Partition podría seleccionar simplemente el primer elemento (A[Lb]). El resto de valores se compararía con el pivot, y sería ubicados a la izquierda o derecha de este como sea apropiado. Sin embargo, hay un caso que puede fracasar miserablemente. Suponga que el arreglo era originalmente ordenado. Partition seleccionaría siempre el valor bajo como pivote y partiría el arreglo con un único

elemento en la partición izquierda, y Ub-Lb elementos en el otro. Cada llamada recursiva a Quicksort podría disminuir el tamaño del arreglo a ordenar en uno. Por lo tanto, *n* llamadas recursivas podrían ser requeridas para que el resultado de la ordenación se un tiempo de $O(n^2)$. Una solución a este problema es una selección randómica del item como pivot. Esto podría convertir en sumamente inverosímil el comportamiento del *peor de los casos.*

Implementación

Typedef ***T*** y el operador de comparación ***compGT*** podrían ser alterados para reflejar los datos guardados en el arreglo. Algunas modificaciones se han hecho al algoritmo Quicksort básico:

- El elemento central es seleccionado como pivot en **particion**. Si la lista es parcialmente ordenada, esta podría ser una buena selección. El peor de los casos (worst-case) podría ocurrir cuando el elemento central provoque que la partición que se invoque sea cada vez más pequeña o más grande.
- Para arreglos cortos, **OrdenacionPorInsercion** es llamada. Debido a la recursividad y otros factores, Quicksort no es un algoritmo eficiente para se rusado en arreglos pequeños. Consecuentemente un arreglo con menos de 12 elementos es ordenado con *insertion sort*. El valor óptimo de cesación no es critico y varía basándose en la calidad del código generado.
- La recursión en cola ocurre cuando la última declaración de una función se llama a sí misma. La recursión en cola puede ser reemplazada por la iteración, resultando en una utilización mejor del espacio de pila. Esto ha dado buenos resultados en la segunda llamada a Quicksort en la Figura 4.2.3
- Luego de que el arreglo es particionando, la partición más pequeña es ordenada primero. Esto resulta en una mejor utilización del espacio de pila, como son particiones pequeñas, es más rápido ordenarlas y dispensarlas.

En el siguiente código fuente, se define una función estándar de librería usualmente implementada con Quicksort. Para esta implementación, llamadas recursivas son reemplazadas por operaciones explícitas de pila. La Tabla 4.2.1 muestra estadísticas de tiempo de ejecución y utilización de pilas antes y después de que se aplicaran los aumentos.

```
#include <stdio.h>
#include <stdlib.h>

typedef int T;          /* tipo de item a ser ordenado */
typedef int tblIndex;   /* tipo de subscript */
```

```
#define compGT(a,b) (a > b)

void OrdenacionPorInsercion(T *a, tblIndex lb, tblIndex ub) {
    T t;
    tblIndex i, j;

   /**************************
      ordena array a[lb..ub]  *
    **************************/
    for (i = lb + 1; i <= ub; i++) {
        t = a[i];

        /* intercambiar elmentos hasta encontrar punto de inserción */

        for (j = i-1; j >= lb && compGT(a[j], t); j--)
            a[j+1] = a[j];

        /* inserta */
        a[j+1] = t;
    }
}

tblIndex particion(T *a, tblIndex lb, tblIndex ub) {
    T t, pivot;
    tblIndex i, j, p;

   /*******************************
      particiona array a[lb..ub]  *
    *******************************/

    /* seleccionar un pivot e intercambiarlo con el 1er elemento */
    p = lb + ((ub - lb)>>1);
    pivot = a[p];
    a[p] = a[lb];

    /* ordena lb+1..ub en base al pivot */
    i = lb+1;
    j = ub;
    while (1) {
        while (i < j && compGT(pivot, a[i])) i++;
        while (j >= i && compGT(a[j], pivot)) j--;
        if (i >= j) break;
        t = a[i];
        a[i] = a[j];
        a[j] = t;
        j--; i++;
    }

    /* pivot se convierte en a[j] */
    a[lb] = a[j];
    a[j] = pivot;

    return j;
}

void OrdenaciónRápida(T *a, tblIndex lb, tblIndex ub) {
    tblIndex m;
```

```
   /*************************
     ordena array a[lb..ub]  *
    *************************/

    while (lb < ub) {

        /* rápidol ordena listas cortas */
        if (ub - lb <= 12) {
            OrdenacionPorInsercion(a, lb, ub);
            return;
        }

        /* parte en dos segmentos */
        m = partition (a, lb, ub);

        /* ordena la partición más pequeña */
        /* para minimizar requerimientos de stack */
        if (m - lb <= ub - m) {
            OrdenaciónRápida(a, lb, m - 1);
            lb = m + 1;
        } else {
            OrdenaciónRápida(a, m + 1, ub);
            ub = m - 1;
        }
    }
}

void llena(T *a, tblIndex lb, tblIndex ub) {
    tblIndex i;
    srand(1);
    for (i = lb; i <= ub; i++) a[i] = rand();
}

int main(int argc, char *argv[]) {
    tblIndex maxnum, lb, ub;
    T *a;

    /* command-line:
     *
      qui maxnum
     *
      qui 2000
      ordenas 2000 registros
     *
     */

    maxnum = atoi(argv[1]);
    lb = 0; ub = maxnum - 1;
    if ((a = malloc(maxnum * sizeof(T))) == 0) {
        fprintf (stderr, "memoria insuficiente \n");
        exit(1);
    }

    llena(a, lb, ub);
    OrdenaciónRápida (a, lb, ub);

    return 0;
```

```
}
```

	tiempo (µs)		tamaño de pila	
cuenta	antes	después	antes	después
16	103	51	540	28
256	1,630	911	912	112
4,096	34,183	20,016	1,980	168
65,536	658,003	470,737	2,436	252

Tabla 4.2.1: Efectos de realce en velocidad y utilización de pila

2.4 Comparación

En esta sección compararemos los algoritmos de ordenación tratados: insertion sort, shell sort y quicksort. Hay varios factores que influyen en la elección de un algoritmo de ordenación:

- Stable sort. Recuerde que una ordenación estable sacará idénticos valores en las mismas posiciones relativas en la salida ordenada. Insertion sort es el único algoritmo cubierto que es estable.
- Space. En una ordenación in-place (en el lugar) no se requiere espacio adicional para cumplir con su tarea. Insertion sort y shell sort son ordenaciones in-place. Quicksort requiere espacio de pila para recursión, y no por esto deja de ser una ordenación in-place. Sin embargo, la cantidad de espacio requerido es considerablemente reducido por tinkering con el algoritmo.
- Tiempo. El tiempo requerido para ordenar un conjunto de datos puede convertirse fácilmente en valores astronómicos (Tabla 1.1), la Tabla 4.2.2 muestra los tiempos relativos para cada método. Los tiempos actuales son descritos más adelante.
- Simplicity (Simplicidad). El número de declaraciones requeridas para cada algoritmo puede observarse en la Tabla 4.2.2. Los algoritmos más simples suelen tener menos errores de programación. El tiempo requerido para ordenar un conjunto de datos randómicamente ordenado se muestra en la Tabla 4.2.3.

Método	Cuentas	Tiempo de ejecución	Peor caso
Insertion sort	9	O(n^2)	O(n^2)
shell sort	17	O(n^1.25)	O(n^1.5)
quicksort	21	O(n lg n)	O(n^2)

Tabla 4.2.2: Comparaciones de Métodos de Ordenación

Cuenta	insertion	shell	quicksort
16	39 µs	45 µs	51 µs
256	4,969 µs	1,230 µs	911 µs
4,096	1.315 s	0.033 s	0.20 s
65,536	416.437 s	1.254 s	0.461 s

Tabla 4.2.3: Tiempos de Ordenación

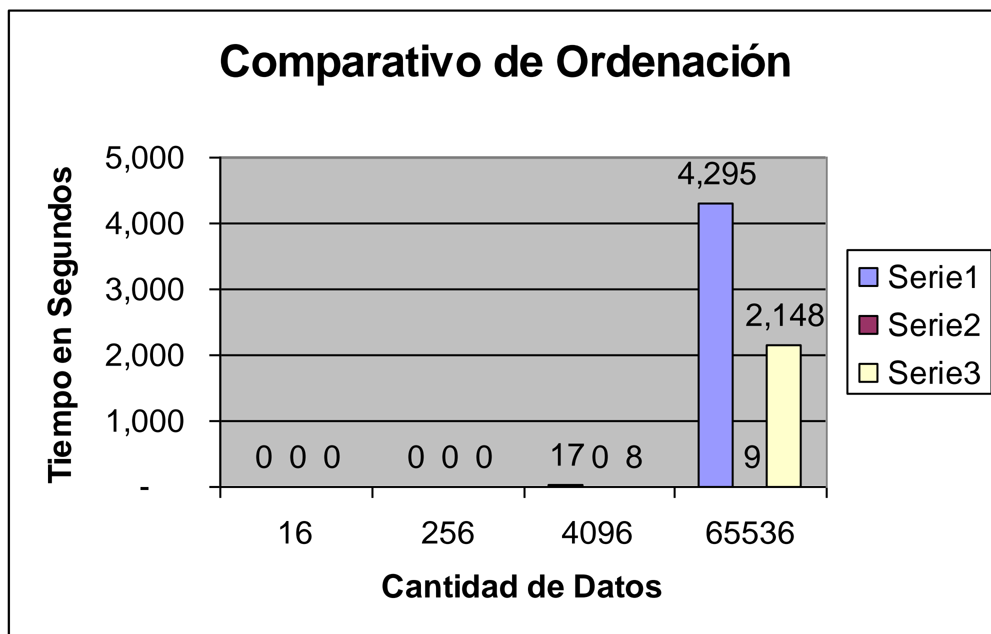


Gráfico 4.2.1: Comparativo en tiempos de Ordenamiento

4.3 Diccionarios

Los diccionarios son estructuras de datos que soportan operaciones de búsqueda, inserción y borrado. Uno de los métodos más efectivos presentados es usar las disperción tables. Típicamente, una simple función es aplicada a el valor para determinar su ubicación en el diccionario. También se incluyen los binary trees (árboles binarios) y red-black trees (árboles red-black). Ambos métodos de árboles utilizan técnicas parecida a los algoritmos de búsqueda binaria para minimizar el número de comparaciones durante las operaciones de búsqueda y actualización en los diccionarios. Finalmente, las skip lists (listas de salto) ilustran un enfoque simple que utiliza números aleatorios para construir un diccionario.

4.3.1 Tablas de Disperción (Hash)

Las tablas de disperción son un simple y efectivo método de implementar diccionarios. El tiempo promedio de búsqueda de elementos es O(1), mientras que en el peor de los casos su tiempo es de O(n). Cormen[1990] y Knuth[1998], ambos contienen excelentes discusiones sobre dispercióning. En caso de que decida leer más material sobre este tema, debería primero conocer alguna terminología. La técnica presentada aquí es en cadena, también conocida como *open hashing*. Una técnica alternativa, conocida como *closed hashing*, no es tratada.

Teoría

Una tabla de disperción es simplemente una arreglo que es direccionado por medio de una función disperción. Por ejemplo, en la Figura 4.3.1 **HashTable** es un arreglo de 8 elementos. Cada elemento es un puntero a una lista enlazada de datos numéricos. La función disperción para este ejemplo simplemente divide los valores para 8, y usa el resto como un índice dentro de la tabla. Estos rinde un número entre 0 y 7. Donde el rango de índices para **HashTable** es de 0 a 7, garantizamos que el índice es válido.

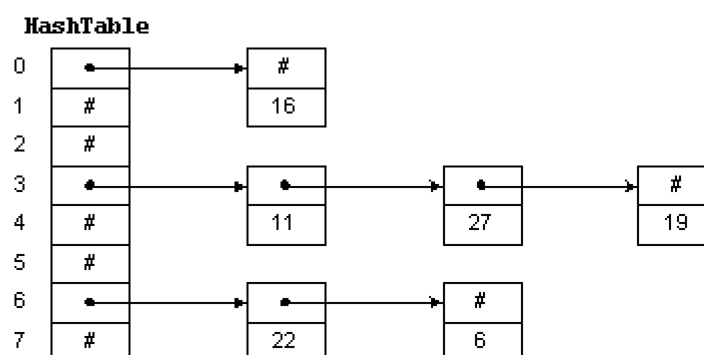


Figura 4.3.1: Una Tabla de disperción

Para insertar un nuevo elemento en la tabla, nosotros aplicamos disperción al valor para determinar donde debe ir en la lista, y entonces insertamos el valor al inicio de la lista. Por ejemplo, para insertar el 11, dividimos 11 para 8 y obtenemos 3 de residuo. Así, 11 va en la lista de entrada a **HashTable[3]**. Para buscar un número, debemos aplicar disperción al número y encadenamos hacia abajo en la lista para ver si se encuentra en la tabla. Para borrar un número, buscamos el número y eliminamos el nodo de la lista enlazada.

Si la función disperción es uniforme, o las claves de los datos están distribuidas igualitariamente dentro de los índices de la tabla has, entonces dispercióning subdivide efectivamente la lista a ser buscada. El peor de los casos ocurre cuando todas las claves de disperción tienen el mismo índice. Consecuentemente, es importante escoger una buena función disperción. Algunos métodos podrían usar valores para claves disperción. Para ilustrar la técnica, asumiremos que *unsigned char* es de 8 bits, *unsigned short* es de 16 bits y *unsigned long* es de 32 bits.

- *Método de la división (tablesize=prime).* Esta técnica fue utilizada en el ejemplo anterior. Un **HashValue**, de 0 a (**HashTableSize** –1), es calculado para dividir los valores de las claves por el tamaño de la tabla de disperción y tomar el residuo. Por ejemplo:

```
Typedef int HashIndexType;

HashIndexType Hash(int Key) {
        Return Key % HashTableSize;
}
```

- Seleccionar un apropiado **HashTableSize** es importante para el desempeño de este método. Por ejemplo, un **HashTableSize** de dos podría producir entonces valores iguales para **Keys**, y valores disperción raros para **Keys** raras. Es una propiedad un tanto incomprensible, como todas las claves podrían aplicar disperción al los valores iguales si estos tienden a ser los mismos. Si **HashTableSize** es un múltiplo de dos, entonces la función disperción simplemente selecciona un subconjunto de las **Key** bits como

índice de la tabla. Para obtener una mayor dispersión randómica, **HashTableSize** podría ser un número primo no muy cerca de un múltiplo de dos.

- *Método de la Multiplicación (tablesize=2^n).* El método de la multiplicación podría ser usado para un **HashTableSize** que es múltiplo de 2. La **Key** es multiplicada, por una constante, y entonces los bits necesarios son extraídos para indexarlos en la tabla. Knuth recomienda usar la parte fraccional del producto de la **Key** y el radio dorado, o . Por ejemplo, asumamos un largo de palabra de 8 bits, el radio dorado (golden ratio) es multiplicado por $2^{\#}$ para obtener 158. El producto de la **Key** de 8 bits y 158 da como resultado un entero de 16 bits. Para un tamaño de tabla de 2^5 los 5 bits más significante de la palabra menos significante son extraídos para valor disperción. Las siguientes definiciones podrían ser utilizadas para un método de multiplicación:

```
/* Indices de 8 –bit */
typedef unsigned char HashIndexType;
static const HashIndexType K=158;

/* Indices de 16-bit  */
typedef unsigned short int HashIndexType;
static const HashIndexType K= 40503;

/* Indices de 32-bit  */
typedef unsigned long int HashIndexType;
static const HashIndexType K=2654435769;

/* w=bitwidth(HashIndexType), size of table= 2**n */
static const int S=w-n;
HashIndexType HashValue = (HashIndexType)(K* Key)>> S;
```

Por ejemplo, si **HashTableSize** es 1025(s^{10}), entonces un índice de 16 bits es suficiente y **S** podría asignar un valor de 16 –10=6, entonces, tenemos:

```
Typedef unsigned short int HashIndexType;

HashIndexType Hash(int Key) {
    Static const HashIndexType K = 40503;
```

```
    Static const int S=6;
    Return (HashIndexType)(K * Key) >> S;
}
```

- *Método de adición de una variable string (tablesize=256).* Para aplicar disperción a una variable de longitud string, cada carácter es adicionado, modulo 256, al total. Un **HashValue**, en el rango 0-255, es calculado.

```
Typedef unsigned char HashIndexType;

HasIndexType Hash(char *str) {
    HashIndexType h = 0;
    While (*str) h += *str++;
    Return h;
}
```

- *Método del or-exclusivo para variables string (tablesize=256).* Este método es similar al método de adición, pero satisfactoriamente diferencia palabras similares y anagramas. Para obtener valores has en el rango 0-255, todos los bytes en la cadena son aplicadas el or-exclusivo. Sin embargo, en el proceso de realizar cada or-exclusivo, un componente randómico es introducido.

```
Typedef unsigned char HashIndexType;
Unsigned char Rand8[256];

HashIndexType Hash(char *str) {
    Unsigned char h = 0;
    While (*str) h = Rand8[h^*str++];
    Return h;
}
```

Rand8 es una tabla de 256 números randómicos únicos de 8 bits. El orden exacto no es crítico. El método del or-exclusivo se basa en la *cryptografía*, y es altamente efectivo.

- *Método del or exclusivo para variables string (tablesize <= 65536).* Si aplicamos disperción a la cadena dos veces, habremos derivado un valor

disperción para una tabla arbitraria de tamaño mayor que 65536. La segunda vez que la cadena es aplicada disperción, uno es añadido al primer carácter. Entonces los dos valores has de 8 bits son concatenados para formar luego el valor has de 16 bits.

```
Typedef unsigned short int HashIndexType;
Unsigned char Rand8[256];

HashIndexType Hash(char *str) {
    HashIndexType h;
    Unsigned char h1,h2;

    If (*str == 0) return 0;
        h1=*str;h2=*str +1;
    str++;
    while (*str) {
            h1=Rand8[h1^*str];
            h2= Rand8[h2^*str];
            str++
    }
    /* h en rango 0…65535 */
    h=((HashIndexType)h1 << 8) | (HashIndexType)h2;
    /* utiliza método de división a escala */
    return h % HashTableSize
}
```

Asumiendo n ítems de datos, el tamaño de la tabla de disperción podría alargarse hasta acomodarse a un número razonable de entradas. Como se observa en la Tabla 4.3.1, un tamaño de tabla pequeño incrementa sustancialmente el tiempo de ejecución para buscar un valor. Una tabla de disperción debería ser observada como una colección de listas enlazadas. Como la tabla tiende a alargarse, el número de listas se incrementa, y el alcance del número de nodos en cada lista decrece. Si el tamaño de la tabla es 1, entonces la tabla es en realidad una simple lista enlazada de longitud n. Asumiendo una función disperción perfecta, el tamaño de tabla 2 es como dos listas de longitud n/2. Si el tamaño de tabla es 100, entonces tenemos 100 listas de longitud n/100. Esto reduce considerablemente la longitud de la lista a ser buscada. Como se mira en la Tabla 4.3.1, hay que ser cuidadosos al escoger el tamaño de una tabla.

tamaño	tiempo	tamaño	tiempo
1	869	128	9
2	432	256	6
4	214	512	4
8	106	1024	4
16	54	2048	3
32	28	4096	3
64	15	8196	3

Tabla 4.3.1: **HashTableSize** vs. Tiempo de Ejecución de Búsqueda (μs), 4096 entradas

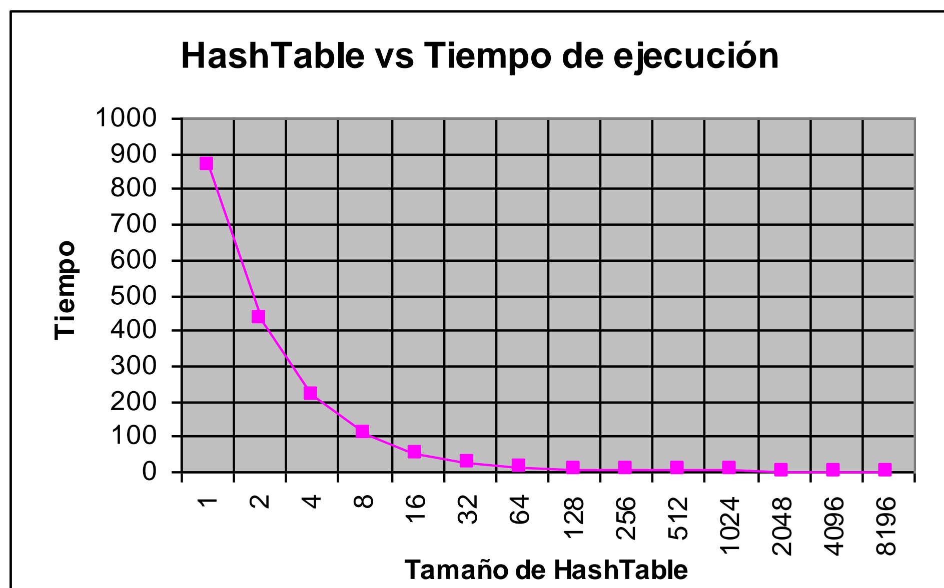


Gráfico 4.3.1: HasTable vs Tiempo de Ejecución

Implementación

Typedef **T** y el operador de comparación **compEQ** podrían ser alterados para reflejar los datos guardados en la tabla. **TamañoTablaHash** debería ser determinado y localizada la **HashTable**. El método de división ha sido utilizado en la función **Disperción**. La función **InsertaNodo** ubica un nuevo nodo e inserta este en la tabla. La función **BorraNodo** borra y libera un nodo desde la tabla. La función **BuscaNodo** localiza en la tabla un valor en particular.

```
#include <stdio.h>
#include <stdlib.h>

typedef int T;                  / tipo de item a ordenar /
typedef int HashTableIndex;     / indice en la tabla de disperción */
#define compEQ(a,b) (a == b)

typedef struct Node_ {
    struct Node_ next;         / nodo siguiente */
    T data;                     /* dato almacenado en nodo */
} Node;


Node **HashTable;
int HashTableSize;

HashTableIndex Hash(T data) {

   /***********************************
      función disperción aplicada a los datos
    ***********************************/

    return (data % HashTableSize);
}

Node *insrtaNodo(T data) {
    Node *p, *p0;
    HashTableIndex bucket;

   /************************************************
      localizar nodo para datos e insertar en tabla
    ************************************************/

    /* insert node at beginning of list */
    bucket = Hash(data);
    if ((p = malloc(sizeof(Node))) == 0) {
        fprintf (stderr, "fuera de memoria (insrtaNodo)\n");
        exit(1);
    }
    p0 = HashTable[bucket];
    HashTable[bucket] = p;
```

```
    p->next = p0;
    p->data = data;
    return p;
}

void BorraNodo(T data) {
    Node *p0, *p;
    HashTableIndex bucket;

   /********************************************
      eliminar nodo que contenga datos en la tabla
    ********************************************/

    /* localizar nodo */
    p0 = 0;
    bucket = Hash(data);
    p = HashTable[bucket];
    while (p && ¡compEQ(p->data, data)) {
        p0 = p;
        p = p->next;
    }
    if (¡p) return;

    /* p  determina el nodo a eliminar, y lo elimina de la lista*/
    if (p0)
        /* no el primer nodo, p0 apunta al nodo anterior */
        p0->next = p->next;
    else
        /* primer nodo en chain */
        HashTable[bucket] = p->next;

    free (p);
}

Node *BuscaNodo (T data) {
    Node *p;

   /******************************
      localizar nodo que contenido en la lista
    ******************************/

    p = HashTable[Hash(data)];
    while (p && ¡compEQ(p->data, data))
        p = p->next;
    return p;
}

int main(int argc, char **argv) {
    int i, *a, maxnum, random;

    /* linea de comandos
     *
      has maxnum HashTable [random]
     *
      has 2000 100
      procesa 2000 registros, tamañotabla=100, núemros secuenciales
      has 2000 100 r
      procesa 2000 registros, tamañotabla=100, núemros randómicos*/
```

```
    maxnum = atoi(argv[1]);
    HashTableSize = atoi(argv[2]);
    random = argc > 3;

    if ((a = malloc(maxnum * sizeof(*a))) == 0) {
        fprintf (stderr, "fuera de memoria (a)\n");
        exit(1);
    }

    if ((HashTable = malloc(HashTableSize * sizeof(Node *))) == 0) {
        fprintf (stderr, "fuera de memoria (HashTable)\n");
        exit(1);
    }

    if (random) { /* randómico */
        /* llenar "a" con un único número randómico */
        for (i = 0; i < maxnum; i++) a[i] = rand();
        printf   ("ran   ht,   %d   items,   %d   HashTable\n",   maxnum,
HashTableSize);
    } else {
        for (i=0; i<maxnum; i++) a[i] = i;
        printf   ("seq   ht,   %d   items,   %d   HashTable\n",   maxnum,
HashTableSize);
    }


    for (i = 0; i < maxnum; i++) {
        insrtaNodo(a[i]);
    }

    for (i = maxnum-1; i >= 0; i--) {
        BuscaNodo(a[i]);
    }

    for (i = maxnum-1; i >= 0; i--) {
        BorraNodo(a[i]);
    }
    return 0;
}
```

4.3.2 Arboles de Búsqueda Binaria

En la introducción, utilizamos el algoritmo de búsqueda binaria para localizar datos almacenados en un arreglo. Este método es muy efectivo, como cada iteración reduce el número de ítems a buscar por one-half. Sin embargo, como los datos son almacenados en una array, las inserciones y eliminaciones no son eficientes. Los arboles de búsqueda binaria almacenan datos en nodos que están enlazados en une estilo árbol. Para una inserción randómica de datos, se utiliza un tiempo de búsqueda de O(lg n). El peor de los casos podría ocurrir cuando datos ordenados son insertados, en este caso el tiempo de búsqueda es de O(n). Mire Cormen para información adicional.

Teoría

Un árbol de búsqueda binaria es un árbol donde cada nodo tiene un hijo izquierdo y derecho. Un hijo, o ambos podrían estar perdidos. La Figura 4.3.2 ilustra un árbol de búsqueda binaria. Asumamos que **Key** representa el valor que toma un nodo, entonces el árbol binario busca en el árbol las siguiente propiedad: todos los hijos de la izquierda del nodo tienen valor menores que Key, y todos los hijos derechos del nodo tendrán valores mayores que Key. El tope del árbol es conocido como raíz, y los nodos que están más hacia abajo del árbol se conocen como niveles. En la Figura 4.3.2 la raíz es el nodo 20 y el nivel tiene los nodos 4,16,37 y 43. La altura del árbol es la longitud del camino entre la raíz y el tope. Para este ejemplo es 2.

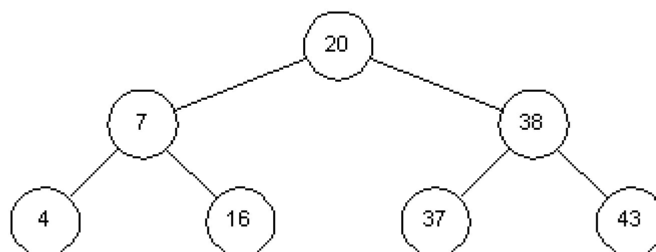


Figura 4.3.2: Un Árbol de Búsqueda Binaria

Para buscar en un árbol un determinado valor, iniciamos en la raíz y trabajamos hacia abajo. Por ejemplo, para localizar el 16, primeramente notamos que 16 < 20 y nos pasamos al hijo izquierdo. La segunda comparación indica que 16>7, entonces nos pasamos al hijo derecho. En una tercera comparación, habremos localizado el nodo.

Cada comparación resulta en una reducción de los ítems a ser inspeccionados. En este respecto, el algoritmo es similar a la búsqueda binaria de un arreglo. Sin embargo, esto es verdad solo si el árbol está balanceado. La Figura 4.3.3 nos muestra otro árbol que contiene los mismo valores, este es más como una lista enlazada, con un tiempo de búsqueda incrementado proporcionalmente al numero de elementos almacenados.

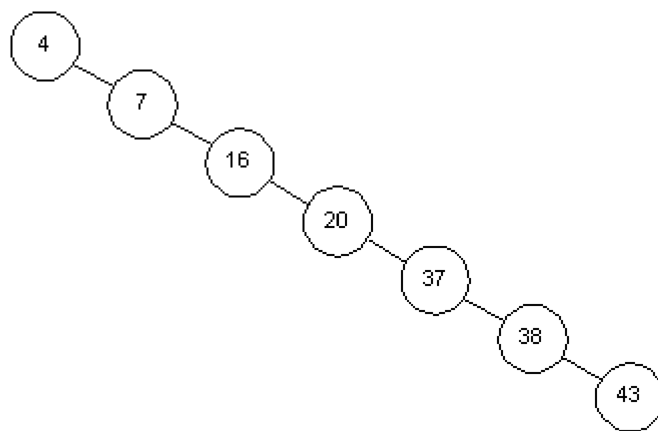


Figura 4.3.3: un Árbol de Búsqueda Binaria no balanceado

Inserción y Eliminación

Examinaremos las inserciones en un árbol de búsqueda binaria para determinar las condiciones que pueden causar un árbol no balanceado. Para insertar el 18 en el árbol de la figura 4.3.2, debemos buscar para ese número. Esto ocasiona que arribemos al nodo 16 con apuntador a ninguna parte. Como18>16, simplemente adicionamos el nodo 18 a la derecha del hijo del nodo 16 (Figura 4.3.4)

Podemos observar como un arboles no balanceado puede ocurrir. Si los datos son presentados en una secuencia ascendente, cada nodo podría ser adicionado a la derecha del nodo previo. Esto podría crear una longitud infinita o lista enlazada. Sin embargo, si los datos están presentados para insertarlos en un orden randómico, entonces un árbol más balanceado es posible.

Las eliminaciones son similares, pero requerimos que un árbol de búsqueda binaria sea mantenido adecuadamente. Por ejemplo, si el nodo 20 en la Figura 4.3.4 es eliminado, podría ser reemplazado por el nodo 37. Est resulta en el árbol ilustrado en la Figura 4.3.5. la rational es para esta selección es como sigue: el sucesor del nodo 20 podría ser escogido porque todos los nodos a la derecha son mayores. Pero, necesitamos seleccionar el menor valor del nodo ubicado a la derecha de 20. Necesitamos realizar una selección más pequeña a la derecha del nodo 20 . para realizar las selecciones, ubicándolas en la parte derecha (nodo 38), y luego en la izquierda del nodo 20. Para realizar una selección, debemos escoger la derecha del nodo 38, luego a la izquierda hasta que llegamos al final del archivo. Este es el sucesor del nodo 20.

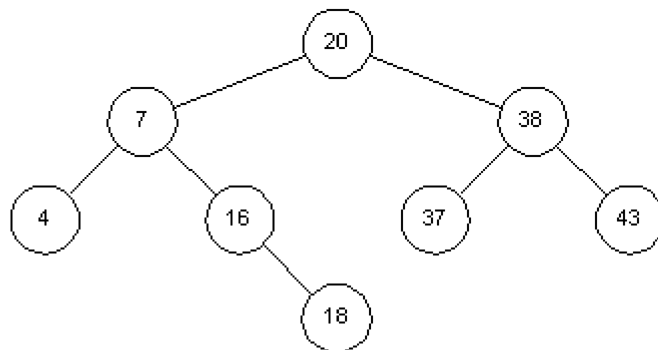


Figura 4.3.4: Árbol Binario Después de Añadir un Nodo

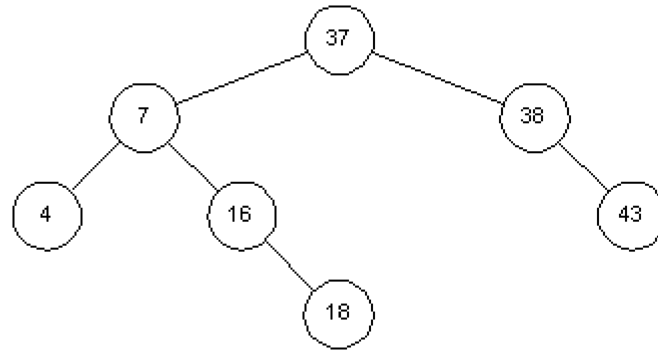


Figura 4.3.5: Árbol Binario Después de Eliminar un Nodo

Implementación

Typedef **T** y el operador de comparación **compEQ** podrían ser alterados para reflejar los datos guardados en el árbol. Cada **Nodo** contiene punteros a **izquierdo**, **derecho** y **parent** que apuntan a cada uno de sus hijos y a su padre. Los datos son almacenados en el campo **data.** El árbol empieza en **raíz**, y es inicialmente **NULL**. La función **InsertNode** localiza un nuevo nodo y lo inserta en el árbol. La función **BorraNodo** borra y libera un nodo del árbol. La función **BuscaNodo** localiza un valor en particular en el árbol.

```
#include <stdio.h>
#include <stdlib.h>

typedef int T;                  /* tipo de item a ordenar */
#define compLT(a,b) (a < b)
#define compEQ(a,b) (a == b)

typedef struct Node_ {
    struct Node_ izquierdo;     /* hijo izquierdo */
    struct Node_ derecho;       /* hijo derecho */
    struct Node_ parent;        /* padre */
    T data;                     /* dato almacenado en nodo */
} Node;

Node raíz = NULL;               /* raíz de árbol binario */

Node *insertNode(T data) {
    Node *x, *current, *parent;

   /**********************************************
      localizar nodo para datos e insertarlo en el árbol
    **********************************************/

    /* buscar el padre de x */
    current = raíz;
    parent = 0;
    while (current) {
        if (compEQ(data, current->data)) return (current);
        parent = current;
        current = compLT(data, current->data) ¿
            current->izquierdo : current->derecho;
    }

    /* configurar nuevo nodo */
    if ((x = malloc (sizede(*x))) == 0) {
        fprintf (stderr, "memoria insuficiente\n");
        exit(1);
    }
    x->data = data;
    x->parent = parent;
```

```
    x->izquierdo = NULL;
    x->derecho = NULL;

    /* insertar x en el árbol */
    if(parent)
        if(compLT(x->data, parent->data))
            parent->izquierdo = x;
        else
            parent->derecho = x;
    else
        raíz = x;

    return(x);
}

void BorraNodo(Node *z) {
    Node *x, *y;

   /*****************************
      elimina nodo z del árbol
    *****************************/

 if (¡z || z == NULL) return;

    /* localizar el sucesor en el árbol */
    if (z->izquierdo == NULL || z->derecho == NULL)
        y = z;
    else {
        y = z->derecho;
        while (y->izquierdo ¡= NULL) y = y->izquierdo;
    }

    /* x es el único hijo de y */
    if (y->izquierdo ¡= NULL)
        x = y->izquierdo;
    else
        x = y->derecho;

    /* remove y from the parent chain */
    if (x) x->parent = y->parent;
    if (y->parent)
        if (y == y->parent->izquierdo)
            y->parent->izquierdo = x;
        else
            y->parent->derecho = x;
    else
        raíz = x;

    /* y nodo que vamos a eliminar */
    /* z dato a elminar */
    /* si z  & y no son iguales, reemplazar z con y*/
    if (y ¡= z) {
        y->izquierdo = z->izquierdo;
        if (y->izquierdo) y->izquierdo->parent = y;
        y->derecho = z->derecho;
        if (y->derecho) y->derecho->parent = y;
        y->parent = z->parent;
        if (z->parent)
```

```
            if (z == z->parent->izquierdo)
                z->parent->izquierdo = y;
            else
                z->parent->derecho = y;
        else
            raíz = y;
        free (z);
    } else {
        free (y);
    }
}

Node *BuscaNodo(T data) {

   /******************************
      localizar nodo que contiene los datos
    ******************************/

    Node *current = raíz;
    while(current ¡= NULL)
        if(compEQ(data, current->data))
            return (current);
        else
            current = compLT(data, current->data) ¿
                current->izquierdo : current->derecho;
    return(0);
}

int main(int argc, char **argv) {
    int i, *a, maxnum, randómicos;

    /* línea de comandos
      bin maxnum randómicos
     *
     *   bin 5000        // 5000 secuenciales
     *   bin 2000 r      // 2000 randómicos
     *
     */
    maxnum = atoi(argv[1]);
    randómicos = argc > 2;

    if ((a = malloc(maxnum * sizede(*a))) == 0) {
        fprintf (stderr, "memoria insuficiente \n");
        exit(1);
    }

    if (randómicos) { /* randómicos */
        /* llenar "a" con un único número randómico */
        for (i = 0; i < maxnum; i++) a[i] = rand();
        printf ("ran bt, %d items\n", maxnum);
    } else {
        for (i=0; i<maxnum; i++) a[i] = i;
        printf ("seq bt, %d items\n", maxnum);
    }


    for (i = 0; i < maxnum; i++) {
        insertNode(a[i]);
```

```
    }

    for (i = maxnum-1; i >= 0; i--) {
        BuscaNodo(a[i]);
    }

    for (i = maxnum-1; i >= 0; i--) {
        BorraNodo(BuscaNodo(a[i]));
    }
    return 0;
}
```

4.3.3 Red-Black Trees (Arboles Rojo-Negro)

Los arboles de búsqueda binaria trabajan mejor cuando están balanceados o el camino entre la raíz y alguna hoja tiene algunos saltos. El algoritmo para arboles red-black es un método para balancear arboles. El nombre deriva del hecho que cada nodo es coloreado rojo o negro, y el color del nodo es el instrumento que determina el balanceo de un árbol. Durante las operaciones de inserción y eliminación, los nodos son rotados para mantener un árbol balanceado. El peor de los casos tiene un tiempo de ejecución de O(lg n).

Esto es talvez, la parte más difícil en el libro. Si usted tiene recelo a los árboles, pase directamente a la siguiente sección. Para lectura adicional consulte Cormen, quien tiene una excelente sección de arboles red-black.

Teoría

Un árbol red-black es un árbol de búsqueda binaria balanceado con las siguientes propiedades:

1. Cada nodo es coloreado de rojo o negro
2. Cada hoja es un nodo **NIL**, y coloreado de negro
3. Si un nodo es rojo, su hijo será negro
4. Todo camino simple desde un nodo a su hoja descendiente contiene el mismo número de nodo negros

El número de nodos negros en un camino desde la raíz a las hojas es conocido como el *black height* del árbol. Estas propiedades generan que algún camino desde la raíz a las hojas no es mayor que el doble de largo que cualquier otro en el mismo árbol. Para ver porque esto es verdad, considere un árbol con un black height de dos. La menor distancia entre la raíz y sus hojas es dos, donde el último nodo es negro. La mayor distancia entre la raíz y sus hojas es de cuatro, donde los nodos son coloreados (raíz a hoja): rojo, negro, rojo, negro. No es posible insertar mas nodos negros porque esto

violaría la propiedad 4, de los requerimientos de los arboles red-black, o dos veces la longitud del camino contenga solo nodos negros. Todas las operaciones en el árbol deben mantener las propiedades listadas arriba. En particular, las operaciones de inserción o eliminación en el árbol deben mantener estas reglas.

Inserción

Para insertar un nodo debemos buscar en el árbol un punto de inserción, y añadir el nodo al árbol. El nuevo nodo reemplaza el nodo existente **NIL** al final del árbol, y tiene dos nodos **NIL** como hijos. En la implementación, un nodo **NIL** es simplemente un puntero a un nodo centinela común que está coloreado de negro. Después de insertar el nuevo nodo, es coloreado de rojo. Entonces sus padres son examinados para determinar si las propiedades del árbol red-black no han sido violadas. Si es necesario, deberemos recolorear el nodo y rotarlo para balancear el árbol.

Para insertar un nodo rojo con dos hijos **NIL**, debemos tener en cuenta de cumplir la propiedad 4 de los arboles red-black. Sin embargo la propiedad 3 puede ser violada. Esta propiedad condiciona que el hijo de un nodo rojo debe ser negro. Mientras los hijos de un nuevo nodo sean negros (ellos son **NIL**), consideramos el caso donde el padre del nuevo nodo es rojo. Insertar un nodo rojo bajo un padre rojo podría violar esta propiedad. Esto son dos casos a considerar:

- *Padre rojo, tío rojo: L*a Figura 4.3.6 ilustra una violación red-red. El nodo X es el nuevo nodo insertado, su padre y su tío son rojos. Una simple recoloración removería la violación red-red. Después de recolorear, el abuelo(nodo B) debe ser chequeado para validarlo, como su padre podría se rojo. Note que este tiene el efecto de propagación de un nodo rojo hacia arriba en el árbol. En complemento, la raíz del arboles marcada negra. Si este es originalmente rojo, entonces esto tiene el efecto de incrementar la black-height del árbol.
- *Padre rojo, tío negro:* La Figura 4.3.7 ilustra una violación red-red, donde el tío es coloreado negro. Aquí los nodos debería ser rotados, con el subárbol

ajustado como se muestra. En este p[unto el algoritmo debe determinar que no existan conflictos red-red y que el tope del subárbol (nodo A) sea de color negro. Note que si el nodo X fuese originalmente un hijo derecho, una rotación a la izquierda podría haber solucionado esto, convirtiendo el nodo a hijo izquierdo.

Cada ajuste realizado mientras se inserta un nodo causa que nosotros recorramos el árbol hacia arriba un paso. Al menos una rotación (2 si el nodo es un hijo derecho) podría ser bueno, como el algoritmo termina en este caso. La técnica de borrado es similar.

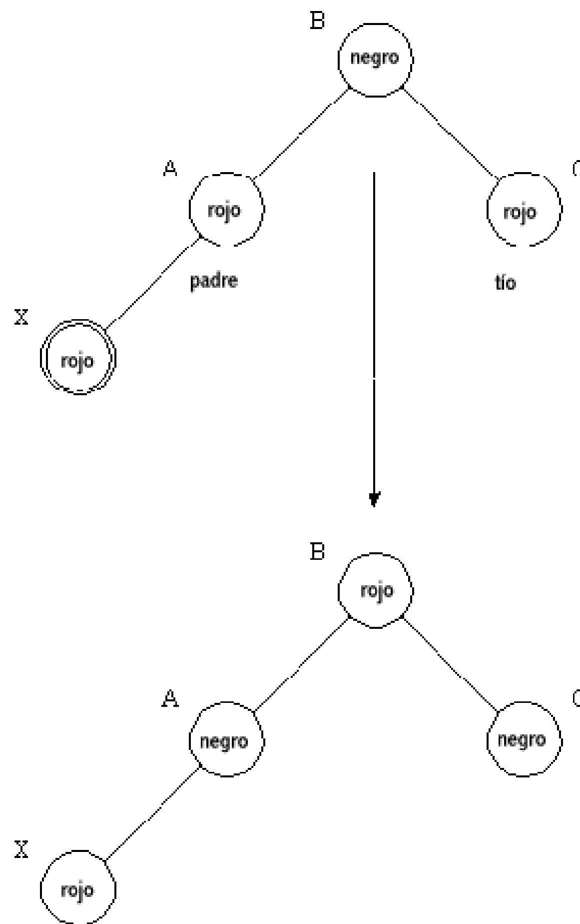


Figura 4.3.6: Inserción – Padre Rojo, Tío Rojo

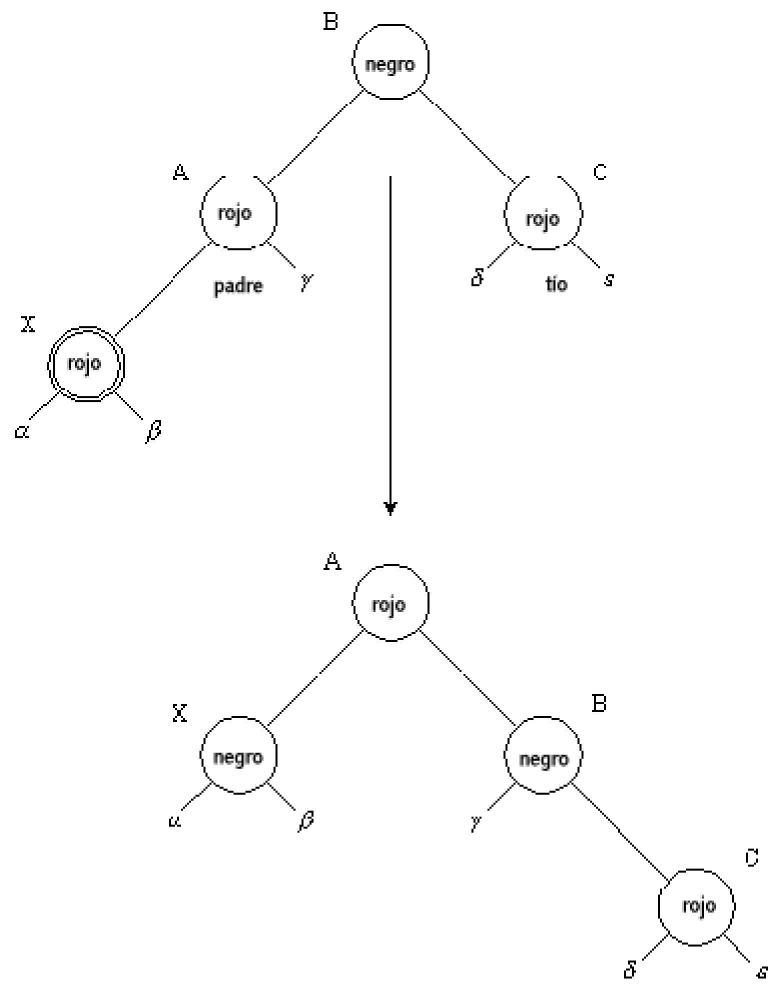


Figura 4.3.7: Inserción – Padre Rojo, Tío Negro

Implementación

Typedef **T** y el operador de comparación **compEQ** podrían ser alterados para reflejar los daa las guardados en el árbol. Cada **Nodo** consiste de punteros **izquierdo, derecho** y **Padre** hacia cada hijo y su padre. El color del nodo es guardado en **color**, u este puede ser **ROJO** o **NEGRO**. Los daa las son almacenados en el campo **data**. A lados los nodos hoja del árbol son nodos **centinela**, para simplificar el código. El árbol comienza en **raíz**, y es inicialmente un nodo **centinela**.

La Función **InsertNodo** localiza un nuevo nodo y lo inserta en el árbol. Subsecuentemente, se llama a **InsertarComponer** para asegurar que las propiedades del árbol rojo-negro se mantienen. Para mantener las propiedades del árbol rojo-negro, se llama a **BorrarBalancear**. La función **fixNodo** localiza en el árbol un valor en particular.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>


typedef int T;                  /* tipo de item a ordenar */
#define compLT(a,b) (a < b)
#define compEQ(a,b) (a == b)

/* Rojo-Negro tree description */
typedef enum { NEGRO, ROJO } NodoColor;

typedef struct Nodo_ {
    struct Nodo_ izquierdo;          / hijo izquierdo */
    struct Nodo_ derecho;         / hijo derecho */
    struct Nodo_ Padre;        / Padre */
    NodoColor color;            /* Nodo color (NEGRO, ROJO) */
    T data;                     /* dato almacenado en nodo */
} Nodo;

#define NIL &sentinel           /* pivot*/
Nodo sentinel = { NIL, NIL, 0, NEGRO, 0};

Nodo raíz = NIL;               / raíz del árbol Rojo-Negro*/

void rotarIzquierdo(Nodo *x) {

   /**************************
      rotar Nodo x a la izquierdo *
```

```
    **************************/

    Nodo *y = x->derecho;

    /* establecer enlace  x->derecho */
    x->derecho = y->izquierdo;
    if (y->izquierdo ¡= NIL) y->izquierdo->Padre = x;

    /* establecer y->Padre */
    if (y ¡= NIL) y->Padre = x->Padre;
    if (x->Padre) {
        if (x == x->Padre->izquierdo)
            x->Padre->izquierdo = y;
        else
            x->Padre->derecho = y;
    } else {
        raíz = y;
    }

    /* enlazar x & y */
    y->izquierdo = x;
    if (x ¡= NIL) x->Padre = y;
}

void rotarDerecho(Nodo *x) {

   /***************************
      rotar Nodo x a la derecha  *
    ***************************/

    Nodo *y = x->izquierdo;

    /* establecer enlace x->izquierdo */
    x->izquierdo = y->derecho;
    if (y->derecho ¡= NIL) y->derecho->Padre = x;

    /* establecer enlace y->Padre */
    if (y ¡= NIL) y->Padre = x->Padre;
    if (x->Padre) {
        if (x == x->Padre->derecho)
            x->Padre->derecho = y;
        else
            x->Padre->izquierdo = y;
    } else {
        raíz = y;
    }

    /* enlazar x & y */
    y->derecho = x;
    if (x ¡= NIL) x->Padre = y;
}

void InsertarComponer(Nodo *x) {

   /*************************************
      mantener árbol Rojo-Negro balanceado
      después de insertar nodo
    *************************************/
```

```
    /* verificar propiedades Rojo-Negro */
    while (x ¡= raíz && x->Padre->color == ROJO) {
        /* we have a violation */
        if (x->Padre == x->Padre->Padre->izquierdo) {
            Nodo *y = x->Padre->Padre->derecho;
            if (y->color == ROJO) {

                /* tío es ROJO */
                x->Padre->color = NEGRO;
                y->color = NEGRO;
                x->Padre->Padre->color = ROJO;
                x = x->Padre->Padre;
            } else {

                /* tío es NEGRO */
                if (x == x->Padre->derecho) {
                    /* hacer x un hijo izquierdo */
                    x = x->Padre;
                    rotarIzquierdo(x);
                }

                /* recolorear y rotar */
                x->Padre->color = NEGRO;
                x->Padre->Padre->color = ROJO;
                rotarDerecho(x->Padre->Padre);
            }
        } else {

            /* sacar imagen del código abajo */
            Nodo *y = x->Padre->Padre->izquierdo;
            if (y->color == ROJO) {

                /* tío es ROJO */
                x->Padre->color = NEGRO;
                y->color = NEGRO;
                x->Padre->Padre->color = ROJO;
                x = x->Padre->Padre;
            } else {

                /* tío es NEGRO */
                if (x == x->Padre->izquierdo) {
                    x = x->Padre;
                    rotarDerecho(x);
                }
                x->Padre->color = NEGRO;
                x->Padre->Padre->color = ROJO;
                rotarIzquierdo(x->Padre->Padre);
            }
        }
    }
    raíz->color = NEGRO;
}

Nodo *insertNodo(T data) {
    Nodo *current, *Padre, *x;

   /************************************************
```

```
      localizar nodo padre en el árbol
    *********************************************/

     /* LOCALIZAR NODO */
    current = raíz;
    Padre = 0;
    while (current ¡= NIL) {
        if (compEQ(data, current->data)) return (current);
        Padre = current;
        current = compLT(data, current->data) ¿
            current->izquierdo : current->derecho;
    }

    /* configurar nuevo nodo */
    if ((x = malloc (sizeof(*x))) == 0) {
        printf ("insuficiente memoria\n");
        exit(1);
    }
    x->data = data;
    x->Padre = Padre;
    x->izquierdo = NIL;
    x->derecho = NIL;
    x->color = ROJO;

    /* insertar nodo en el árbol */
    if(Padre) {
        if(compLT(data, Padre->data))
            Padre->izquierdo = x;
        else
            Padre->derecho = x;
    } else {
        raíz = x;
    }

    InsertarComponer(x);
    return(x);
}

void BorrarBalancear(Nodo *x) {

   /**************************************
*  mantener balanceado el árbol luego de eliminar nodo
    ************************************/

    while (x ¡= raíz && x->color == NEGRO) {
        if (x == x->Padre->izquierdo) {
            Nodo *w = x->Padre->derecho;
            if (w->color == ROJO) {
                w->color = NEGRO;
                x->Padre->color = ROJO;
                rotarIzquierdo (x->Padre);
                w = x->Padre->derecho;
            }
            if (w->izquierdo->color == NEGRO && w->derecho->color ==
NEGRO) {
                w->color = ROJO;
                x = x->Padre;
            } else {
```

```
                if (w->derecho->color == NEGRO) {
                    w->izquierdo->color = NEGRO;
                    w->color = ROJO;
                    rotarDerecho (w);
                    w = x->Padre->derecho;
                }
                w->color = x->Padre->color;
                x->Padre->color = NEGRO;
                w->derecho->color = NEGRO;
                rotarIzquierdo (x->Padre);
                x = raíz;
            }
        } else {
            Nodo *w = x->Padre->izquierdo;
            if (w->color == ROJO) {
                w->color = NEGRO;
                x->Padre->color = ROJO;
                rotarDerecho (x->Padre);
                w = x->Padre->izquierdo;
            }
            if (w->derecho->color == NEGRO && w->izquierdo->color ==
NEGRO) {
                w->color = ROJO;
                x = x->Padre;
            } else {
                if (w->izquierdo->color == NEGRO) {
                    w->derecho->color = NEGRO;
                    w->color = ROJO;
                    rotarIzquierdo (w);
                    w = x->Padre->izquierdo;
                }
                w->color = x->Padre->color;
                x->Padre->color = NEGRO;
                w->izquierdo->color = NEGRO;
                rotarDerecho (x->Padre);
                x = raíz;
            }
        }
    }
    x->color = NEGRO;
}

void deleteNodo(Nodo *z) {
    Nodo *x, *y;

   /****************************
      eliminar nodo del árbol
    ****************************/

    if (¡z || z == NIL) return;


    if (z->izquierdo == NIL || z->derecho == NIL) {
        /* y tiene un nodo NIL como hijo */
        y = z;
    } else {
       /* localizar un sucesor en el árbol con un nodo NIL como hijo */
        y = z->derecho;
```

```
        while (y->izquierdo ¡= NIL) y = y->izquierdo;
    }

    /* x is y's only child */
    if (y->izquierdo ¡= NIL)
        x = y->izquierdo;
    else
        x = y->derecho;

x->Padre = y->Padre;
    if (y->Padre)
        if (y == y->Padre->izquierdo)
            y->Padre->izquierdo = x;
        else
            y->Padre->derecho = x;
    else
        raíz = x;

    if (y ¡= z) z->data = y->data;


    if (y->color == NEGRO)
        BorrarBalancear (x);

    free (y);
}

Nodo *findNodo(T data) {

   /******************************
      localizar nodo
    ******************************/

    Nodo *current = raíz;
    while(current ¡= NIL)
        if(compEQ(data, current->data))
            return (current);
        else
            current = compLT (data, current->data) ¿
                current->izquierdo : current->derecho;
    return(0);
}

void main(int argc, char **argv) {
    int a, maxnum, ct;
    Nodo *t;

    /* línea de comandos:
     *
      rbt maxnum
     *
      rbt 2000
      procesa 2000 registros
     *
     */

    maxnum = aa lai(argv[1]);
```

```
    for (ct = maxnum; ct; ct--) {
        a = rand() % 9 + 1;
        if ((t = findNodo(a)) ¡= NULL) {
            deleteNodo(t);
        } else {
            insertNodo(a);
        }
    }
}
```

4.3.4 Skip Lists (Listas de Salto)

Las listas de salto son listas enlazadas que le permiten saltar al nodo correcto. Hasta ahora el rendimiento del inherente cuello de botella en un recorrido secuencial es evitado, mientras que para la inserción y eliminación queda relativamente eficiente. Tiempo de ejecución es de O(lg n). El peor de los casos tiene un tiempo de O(n), pero es extremadamente improbable. Una excelente referencia para listas de salto es Pugh.

Teoría

El esquema de indexación empleado en las listas de salto es similar en naturaleza al método usado para localizar nombres en l un libro de direcciones. Para localizar un nombre, usted apunta al indicador que representa la primera letra de la entrada a buscar. En la Figura 4.3.8, por ejemplo, el tope de la lista representa una lista enlazada simple sin tabs (índices de lista). Adicionando tabs (figura de la mitad), facilitamos la búsqueda. En este caso, los punteros de nivel 1 son recorridos. Entonces el segmento correcto de la lista es encontrado, el puntero del nivel 0 es recorrido para localizar la entrada específica.

El esquema de indexación puede ser extendido como se muestra en la figura última, donde ahora tenemos un índice a los índices. Para localizar un ítem, el puntero de nivel 2 es recorrido hasta que el segmento correcto de la lista es identificado. Subsecuentemente, los punteros de nivel 0 y nivel 1 son recorridos.

Durante la inserción el un número de apuntadores requeridos para un nuevo nodo debe ser determinado. Esto es fácil de resolver usando la técnica de probabilidad. Un generador de números randómicos es utilizado para este propósito (to loss a computer coin). Cuando insertamos un nuevo nodo, volvemos a generar un número para determinar si se trata de un nivel 1. Si acertamos, volvemos a generar otro número para determinar si el nodo podría ser de nivel 2. Otra vez acertamos y generamos nuevamente otro número para determinar si el nodo puede ser de nivel 3. Este proceso se repite hasta que Ud. Pierda o fracase. Si un único nivel es implementado, la estructura de datos es

una simple lista enlazada con un tiempo de búsqueda de O(n). Sin embargo, si suficientes niveles son implementados, la lista enlazada puede ser vista como un árbol con la raíz en el nivel más alto, y con un tiempo de búsqueda de O(lg n).

El algoritmo para listas de saltos tiene un componente probabilístico, y es este el que nos permite ganar bonos en el tiempo ejecución requerido. Sin embargo, esos bonos nos quitan derechos en circunstancias normales. Por ejemplo, para buscar en una lista que contenga 1000 ítems, la probabilidad de el tiempo de búsqueda nos lleve 5 veces el promedio es cerca de 1 en 1,000,000,000,000,000,000[5].

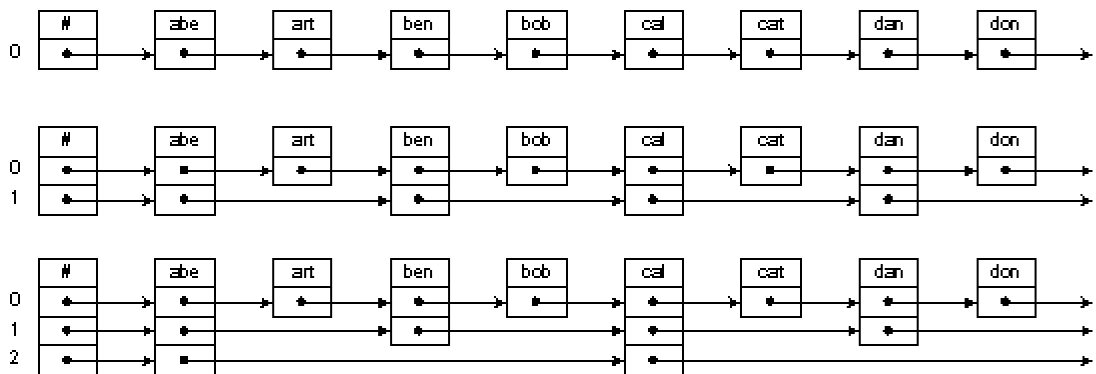


Figura 4.3.8: Construcción de una Lista de Saltos

Implementación

Typedef **T** y el operador de comparación **compEQ** podrían ser alterados para reflejar los datos guardados en la lista. Adicionalmente **MAXIMONIVEL** debería estar basado en el tamaño máximo del conjunto de datos.

Para inicializar, **InicializaLista** es llamado. La cabecera de la lista es localizada e inicializada. Para indicar una lista vacía, todos los valores son configurados y apuntados a la cabecera. La función **InsertarNodo** localiza un nuevo nodo, buscando su lugar correcto de inserción, y lo inserta en la lista. Mientras está buscándolo, el arreglo **Actualiza** mantiene punteros a los nodos de más alto nivel encontrados. Esta información es subsecuentemente utilizada para establecer enlaces correctos para los nuevos nodos insertados. El **newLevel** es determinado utilizando un generador de números randómicos, y la localización del nodo. El siguiente enlace es entonces establecido utilizando información del arreglo **Actualiza**. La función **BorraNodo** elimina y libera un nodo, y es implementado de una manera similar. La función **BuscarNodo** localiza un valor en particular dentro de la lista.

```
#include <stdio.h>
#include <stdlib.h>

typedef int T;                  /* tipo de item a ordenar */
#define compLT(a,b) (a < b)
#define compEQ(a,b) (a == b)

/* rang de niveles entre (0 .. MAXIMONIVEL) */
#define MAXIMONIVEL 15

typedef struct Node_ {
    T data;                     /*datos del usuario */
    struct Node_ *forward[1];   /* punteros a la lista de saltos*/
} Node;

typedef struct {
    Node *hdr;                  /* cabeza de la lista */
    int listLevel;              /* nivel actual de la lista */
} SkipList;

SkipList list;                  /* información de la lista de salto */

#define NIL list.hdr

Node *InsertarNodo(T data) {
```

```
    int i, newLevel;
    Node *Actualiza[MAXIMONIVEL+1];
    Node *x;

   /*********************************************
      localizar nodo para insertar datos
    *********************************************/

    /* find where data belongs */
    x = list.hdr;
    for (i = list.listLevel; i >= 0; i--) {
        while (x->forward[i] != NIL
          && compLT(x->forward[i]->data, data))
            x = x->forward[i];
        Actualiza[i] = x;
    }
    x = x->forward[0];
    if (x != NIL && compEQ(x->data, data)) return(x);

    /* determinar nivel*/
    newLevel = 0;
    while (rand() < RAND_MAX/2) newLevel++;
    if (newLevel > MAXIMONIVEL) newLevel = MAXIMONIVEL;

    if (newLevel > list.listLevel) {
        for (i = list.listLevel + 1; i <= newLevel; i++)
            Actualiza[i] = NIL;
        list.listLevel = newLevel;
    }

    /* crear nuevo nodo */
    if ((x = malloc(sizeof(Node) +
      newLevel*sizeof(Node *))) == 0) {
        printf ("memoria insuficiente\n");
        exit(1);
    }
    x->data = data;

    /* Actualiza enlaces */
    for (i = 0; i <= newLevel; i++) {
        x->forward[i] = Actualiza[i]->forward[i];
        Actualiza[i]->forward[i] = x;
    }
    return(x);
}

void BorraNodo(T data) {
    int i;
    Node *Actualiza[MAXIMONIVEL+1], *x;

   /*****************************************
      eliminar nodo de la lista
    *****************************************/

    /* localizar donde comienza la información */
    x = list.hdr;
    for (i = list.listLevel; i >= 0; i--) {
        while (x->forward[i] != NIL
```

```
          && compLT(x->forward[i]->data, data))
            x = x->forward[i];
        Actualiza[i] = x;
    }
    x = x->forward[0];
    if (x == NIL || !compEQ(x->data, data)) return;

    /* ajustar punteros */
    for (i = 0; i <= list.listLevel; i++) {
        if (Actualiza[i]->forward[i] != x) break;
        Actualiza[i]->forward[i] = x->forward[i];
    }

    free (x);

    /* ajuastar nivel de cabeza */
    while ((list.listLevel > 0)
    && (list.hdr->forward[list.listLevel] == NIL))
        list.listLevel--;
}

Node *BuscarNodo(T data) {
    int i;
    Node *x = list.hdr;

   /********************************
      localizar nodo
    ********************************/

    for (i = list.listLevel; i >= 0; i--) {
        while (x->forward[i] != NIL
          && compLT(x->forward[i]->data, data))
            x = x->forward[i];
    }
    x = x->forward[0];
    if (x != NIL && compEQ(x->data, data)) return (x);
    return(0);
}

void InicializaLista() {
    int i;

   /***************************
      inicializar lista
    ***************************/

    if ((list.hdr = malloc(sizeof(Node) + MAXIMONIVEL*sizeof(Node *)))
== 0) {
        printf ("memoria insuficiente\n");
        exit(1);
    }
    for (i = 0; i <= MAXIMONIVEL; i++)
        list.hdr->forward[i] = NIL;
    list.listLevel = 0;
}

int main(int argc, char **argv) {
    int i, *a, maxnum, random;
```

```
    /* línea de comandos:
     *
     *   skl maxnum [random]
     *
     *   skl 2000
     *       procesa 2000 registros secuenciales
     *   skl 4000 r
     *       procesa 4000 registros randómicos
     *
     */

    maxnum = atoi(argv[1]);
    random = argc > 2;

    InicializaLista();

    if ((a = malloc(maxnum * sizeof(*a))) == 0) {
        fprintf (stderr, "memoria insuficiente \n");
        exit(1);
    }

    if (random) {
        /* llenar "a" con un único valor randómico */
        for (i = 0; i < maxnum; i++) a[i] = rand();
        printf ("ran, %d items\n", maxnum);
    } else {
        for (i = 0; i < maxnum; i++) a[i] = i;
        printf ("seq, %d items\n", maxnum);
    }

    for (i = 0; i < maxnum; i++) {
        InsertarNodo(a[i]);
    }

    for (i = maxnum-1; i >= 0; i--) {
        BuscarNodo(a[i]);
    }

    for (i = maxnum-1; i >= 0; i--) {
        BorraNodo(a[i]);
    }
    return 0;
}
```

4.3.5 Comparación

Debemos analizar algunas maneras de construir diccionarios: tablas de disperción, arboles de búsqueda binaria no balanceados, arboles red-black y listas de salto. Estos son algunos factores que influyen en escoger un algoritmo:

- Salida Ordenada. Si la salida ordenada es requerida, entonces las tablas de disperción no son la mejor alternativa. Las entradas son guardadas en una tabla basada en valores dispersos, sin orden. Para arboles binarios, la historia es diferente. En un árbol ordenado su recorrido podría producir una lista ordenada. Por ejemplo:

```
Void WalkTree(Node *P) {
    If (P== NIL) return;
    WlakTree (P->Left);
    /* examine P -> Data here */
    WalkTree(P->Right;
}
WalkTree(Root);
```

- Para examinar los nodos en una lista de salto ordenada, simplemente encadene a través de los punteros de nivel 0. Por ejemplo:

```
Node *P = List.Ndr->Forward[0];
While(P ¡= NIL) {
    /* examine P->Data here */
    P = P->Forward[0];
}
```

- Espacio. La cantidad de memoria requerida para almacenar un valor podría ser minimizada. Es especialmente cierto si algunos nodos pequeños pueden ser reubicados.
- Para tablas de disperción, solo un puntero hacia delante es requerido. Además, la tabla de disperción en si puede ser reubicada.
- Para arboles red-black, cada nodo tiene punteros left, right y parent. Además, el color de cada nodo puede ser cambiado. Sin embargo esto requiere de un solo bit,

mas espacio podría ser ubicado para que el tamaño de la estructura es alineada apropiadamente. Hasta ahora, cada nodo en un árbol red-black requiere espacio para 3-4 puntero.

- Para listas de salto, cada nodo tiene un puntero de nivel 0 hacia delante. La probabilidad de tener un puntero de nivel 1 es de ½. La probabilidad de tener un puntero de nivel 2 es de ¼. En general, el número de punteros hacia delante por nodo es

$$N= 1 = ½ + ¼…=2$$

  - Tiempo. El algoritmo puede ser eficiente. Esto es especialmente cierto si el largo del conjunto de datos es conocido. La tabla 3.2 compara los tiempos de búsqueda de cada algoritmo. Nótese que en el peor de los casos su comportamiento para tablas de disperción y listas de salto es extremadamente inesperado. Las pruebas actuales de tiempo son descritas a continuación.
  - Simplicidad. Si el algoritmo es pequeño y fácil de entender, algunos cambios pueden ser realizados. Esto no solo hacen su vida más fácil, pero la confianza del mantenimiento del programa con tareas que puedan reparar podría apreciar algún esfuerzo suyo en esta área. El número de declaraciones requeridas para cada algoritmo es listada en la siguiente tabla

método	declaraciones	tiempo de ejecución	peor de los casos
tabla de disperción	26	O(1)	O(n)
arboles no balanceados	41	O(Lg n)	O(n)
arboles red-black	120	O(lg n)	O(lg n)
listas de salto	55	O(Lg n)	O(n)

Tabla 4.3.2: Comparación de Diccionarios

El tiempo de ejecución para inserción, búsqueda y eliminación en una base de datos de 65,536(s^{16}) elementos ingresados randómicamente puede observarse en la tabla 4.3.3. para esta prueba la tabla de disperción tiene un tamaño de 10,009 y 16 niveles de

índices para lista de salto. Mientras esto tiene algunas variaciones para los cuatro métodos, existen otras consideraciones a tomar en la elección de un algoritmo.

método	inserción	búsqueda	eliminación
tabla de disperción	18	8	10
arboles no balanceados	37	17	26
arboles red-black	40	16	37
listas de saltos	48	31	35

Tabla 4.3.3: Tiempos de Ejecución (μs) para 65536 ítems, ingresados randómicamente

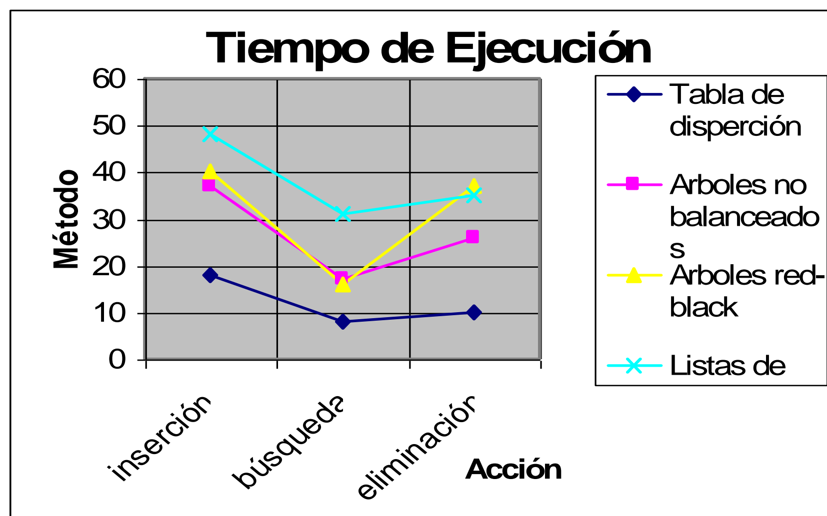


Gráfico 4.313: Tiempos de Ejecución (μs) para 65536 ítems, ingresados randómicamente

La tabla 4.3.4 ilustra los tiempos de búsqueda para dos conjuntos de datos: un conjunto randómico, donde todos los valores son únicos, y una conjunto ordenado, donde los valores son en orden ascendente. La entrada ordenada crea un escenario worst-case (peor de los casos) par los algoritmos de árboles no balanceados, como el fin del árbol viene a ser una lista enlazada. Los tiempos muestran una simple operación de búsqueda. Si desearíamos buscar por todos los valores de una base de datos de 65536 ítems, el algoritmo del árbol red-black podría tomar 0.6 segundos, mientras que un algoritmo del árbol no balanceado podría tomar una hora.

	cuenta	tabla de disperción	arboles no balanceados	arboles red-black	listas de salto
entrada randómica	16	4	3	2	5
	256	3	4	4	9
	4,096	3	7	6	12
	65,536	8	17	16	31
entrada ordenada	16	3	4	2	4
	256	3	47	4	7
	4,096	3	1,033	6	11
	65,536	7	55,019	9	15

Figura 4.3.4: Tiempos de búsqueda (µs)

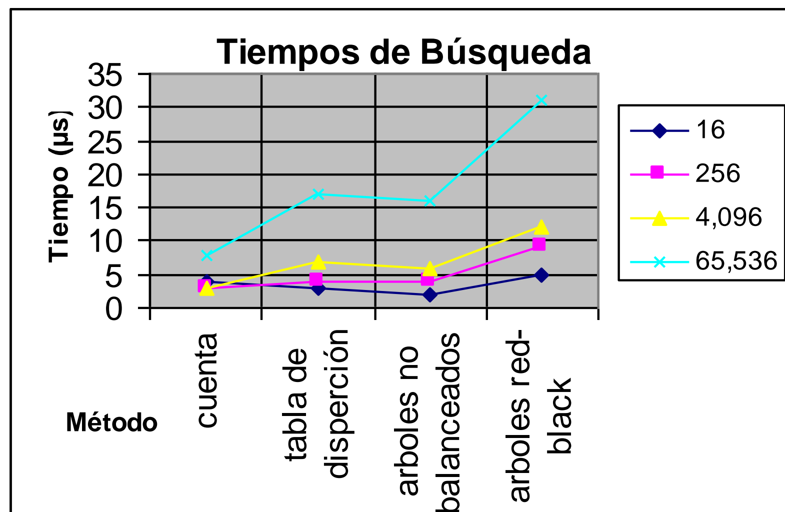


Gráfico 4.3.2: Tiempos de búsqueda (µs)

4.4 Archivos Muy Largos

Los algoritmos anteriores asumen que todos los datos residen en memoria. Sin embargo, muchas veces tenemos bases de datos muy largas, y métodos alternativos son requeridos. En esta sección, examinaremos técnicas para ordenar (ordenaciones externas) e implementar diccionarios (B-trees) para archivos muy largos.

4.4.1 Ordenación Externa

Un método para ordenar un archivo es cargar el archivo en memoria, ordenar los datos en memoria, entonces escribir los resultados. Cuando el archivo no puede ser cargado en memoria debido a limitación de recursos, una ordenación externa es aplicada. Podríamos implementar una ordenación externa utilizando selecciones de reemplazo para establecer las corridas iniciales, seguido de una ordenación combinada multifase para combinar las corridas en un solo archivo ordenado. Se recomienda consultar Knuth para lectura adicional, ya que muchos detalles tuvieron que ser omitidos.

Teoría

Para claridad, asumimos que los datos están en una o más carretes de cintas magnéticas. La Figura 4.4.1 ilustra una combinación multifase de 3 vías. Inicialmente, en l fase A, todos los datos están en las cintas T1 y T2. Asumimos que el comienzo de cada cinta es a al fina del grupo. Estas son dos corridas secuenciales de datos en T1: 4-8, y 6-7. La cinta T2 tiene una corrida: 5-9. Al llegar a la fase B, combinamos la primera corrida de las cintas T1 (4-8) y T2(5-9) en una corrida larga en la cinta T3 (4-5-8-9). La fase C simplemente renombra las cintas, sin embargo podríamos repetir la combinación otra vez. En la fase D repetimos la combinación, con la salida final en la cinta T3.

Fase	T1	T2	T3
A	7 6 8 4	9 5	
B	7 6		9 8 5 4
C	9 8 5 4	7 6	
D			9 8 7 6 5 4

Figura 4.4.1: Ordenación por Combinación

Algunos detalles interesantes tuvieron que ser omitidos en la ilustración anterior. Por ejemplo, donde fueron creadas las corridas iniciales?. Y, puede Ud. Notar que la combinación es perfecta, si corridas extras en alguna cinta?. Mas adelante explicaré el método utilizado para construir corridas iniciales, dejándome disminuirlos para un bit.

En 1202, Leonardo Fibonacci presentó el siguiente ejercicio en su Liber Abbaci (Libro del Abaco): “Como algunos pares de conejos pueden ser reproducidos desde un solo par en el período de un año?”. Nosotros asumimos que cada par produce un nuevo par de descendientes cada mes, cada par llega a ser fértil a la edad de un mes, y que los conejos nunca mueren después de un mes, entonces serán dos pares de conejos, después de dos más habrá 3, el mes siguiente el par original y el par nacido durante el primer mes tendrán un nuevo par y serán 5 en total, y así sucesivamente. Esta serie, donde cada número es la suma del dos números anteriores, es conocida como la serie de Fibonacci:

0,1,1,2,3,5,8,13,21,34,55,89,…

Curiosamente, la serie de Fibonacci ha encontrado un amplio campo de acción para todo, desde el arreglo de flores en las plantas al estudio de la eficiencia del algoritmo de Euclides. Y como Ud. Podría sospechar, la serie de Fibonacci a sido sometida a corridas iniciales establecidas para ordenamiento externo

Retomando el hecho de que teníamos uno en la corrida de la cinta T2, y 2 en la de la cinta T1. Note que los números {1,2} son dos números secuenciales en la serie de Fibonacci. Después de la primera combinación, tenemos una corrida en T1 y otra en T2. Note que los números {1,1} son dos números secuenciales en la serie de Fibonacci. Podemos predecir, de hecho, que si tenemos 13 corridas en T2, y 21 en T1 {13,21}, habremos tenido antes 8 corridas en T1 y 13 en T3 {8,13} después de una pasada. Pasadas sucesivas podrían dar resultados de {5,8], {3,5},{2,3},{1,1} y {0,1}, para un total de 7 pasadas. Est e arreglo es ideal y podría resultar en el mínimo número de pasadas. Podrían los datos estar actualmente en una cinta, este es un gran almacenamiento, como cintas pueden se montadas y rebobinadas para cada pasada. Para mas de 2 cintas, los números de Fibonacci de alto orden son utilizados.

Inicialmente, todos los datos están en una cinta. La cinta es leía , y las corridas son distribuidas a otras cintas en el sistema. Después de que la corrida inicial es creada, ellas son combinadas como se describe anteriormente. Un método que podemos utilizar para crear la corrida inicial es leer un lote de registros en memoria, ordenarlos y escribirlos. Este proceso puede continuar hasta que hayamos terminado las entradas de la cinta. Una alternativa, es el algoritmo de selección y reemplazo, dejado para corridas largas. Un buffer es localizado en memoria para actuar como un lugar de almacenamiento para algunos registros. Inicialmente, el buffer es llenado. Entonces, los siguientes pasos son repetidos hasta que la entrada se acabe:

- Seleccione el registro cuya clave mas pequeña se a menor o igual que la clave del ultimo registro escrito.
- Si todas las claves son pequeñas que la clave del ultimo registro escrito, tenemos que alargar el fin de la corrida. Seleccione el registro con la clave más pequeña como el primer registro de la siguiente corrida.

- Escriba el registro seleccionado
- Reemplace el registro seleccionado con el nuevo registro ingresado.

La Figura 4.4.2 ilustra la selección por reemplazo para un archivo pequeño. El comienzo de una archivo está en la derecha de cada bloque. Para guardar cosas simples, debemos localizar un buffer de 2 registros. Típicamente, un buffer podría almacenar cientos de registros. Cargamos el buffer en el paso B, y escribimos el registro con la clave más pequeña que sea mayor o igual que 6 en el paso D. Esta clave es 7. Después de escribir el 7, reemplazamos esta con la clave 4. Este proceso se repite hasta que el paso F, donde nuestra ultima clave escrita fue 8, y todas las claves son menores que 8. En este punto, terminamos la corrida e iniciamos otra.

Paso	Entrada	Buffer	Salida
A	5-3-4-8-6-7		
B	5-3-4-8	6-7	
C	5-3-4	8-7	6
D	5-3	8-4	7-6
E	5	3-4	8-7-6
F		5-4	3 | 8-7-6
G		5	4-3 | 8-7-6
H			5-4-3 | 8-7-6

Figura 4.4.2: Selección por Reemplazo

Esta estrategia simplemente utiliza un buffer intermedio para almacenar hasta el momento apropiado de su salida. Si se utiliza números randómicos como entrada, la promedio de longitud de una corrida es dos veces la longitud del buffer. Sin embargo, si los datos están algo ordenados, la corrida puede ser extremadamente larga. Hasta ahora este método es más efectivo que hacer ordenamientos parciales.

Cuando seleccionamos el siguiente registro de salida, necesitamos encontrar el valor más pequeño que sea menor o igual que el último valor escrito. . sin embargo, cuando el buffer almacena millones de registros, el tiempo de ejecución viene a ser

prohibitivo., un método alternativo es utilizar una estructura de árboles binarios, de modo que solamente comparemos **lg n** ítems.

Implementación

La función **Corrida** llama a **LeeReg** para leer el siguiente registro. La función **LeeReg** emplea el algoritmo de selección por reemplazo (utilizando un árbol binario) para buscar el siguiente registro, y la función **Corrida** distribuye los registros en una distribución Fibonacci. Si el número de corridas no es un perfecto número Fibonacci, una corrida falsa (dummy run) es simulada al inicio de cada **file** (archivo). La función **MergSort** es entonces llamada para hacer un ordenamiento emergente polifase en la corrida.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/* plantilla para trabajar archivos*/
#define FNAME "_sort%03d.dat"
#define LNAME 13

/* operadores de comparación */
#define compLT(x,y) (x < y)
#define compGT(x,y) (x > y)

/* definir registro a ser ordenado */
#define LRECL 100
typedef int keyType;
typedef struct recTypeTag {
    keyType key;
#if LRECL
        char data[LRECL-sizeof(keyType)];       /* otros campos */
    #endif
} recType;

/******************************
implementación independiente
 ******************************/

typedef enum {false, true} bool;

typedef struct tmpFileTag {
    FILE *fp;                    /* puntero a archivo */
    char name[LNAME];            /* nombre archivo */
    recType rec;                 /* último registro leído */
    int dummy;                   /* númeor de corrida dummy*/
    bool eof;                    /* bandera de fin de archivo */
    bool eor;                    /* bandera fin de corrida */
    bool valid;                  /* true si ret es valido */
    int fib;                     /* número fibonacci ideal */
} tmpFileType;
```

```
static tmpFileType **file;      /* arreglo de archivos para TMPs*/
static int nTmpFiles;           /* número de temporales */
static char *ifName;            /* archivo de entrada */
static char *ofName;            /* archivo de salida */

static int level;               /* nivel de corrida */
static int nNodes;              /* númeor de nodos para selección */

void BorrarArchivosTMP(void) {
    int i;

    /* liberar recursos */
    if (file) {
        for (i = 0; i < nTmpFiles; i++) {
            if (file[i]) {
                if (file[i]->fp) fclose(file[i]->fp);
                if (*file[i]->name) remove(file[i]->name);
                free (file[i]);
            }
        }
        free (file);
    }
}

void termTmpFiles(int rc) {

    /* limpiar archivos */
    remove(ofName);
    if (rc == 0) {
        int fileT;

        /* file[T] contiene resultados */
        fileT = nTmpFiles - 1;
        fclose(file[fileT]->fp); file[fileT]->fp = NULL;
        if (rename(file[fileT]->name, ofName)) {
            perror("io1");
            BorrarArchivosTMP();
            exit(1);
        }
        *file[fileT]->name = 0;
    }
    BorrarArchivosTMP();
}

void LimpiarSalida(int rc) {

    /* limpia tmp y termina con Exit */
    termTmpFiles(rc);
    exit(rc);
}

void *asegurarMalloc(size_t size) {
    void *p;

    /* asegurar memoria e inicializarla con cero */
    if ((p = calloc(1, size)) == NULL) {
        printf("error: malloc falló, size = %d\n", size);
```

```
        LimpiarSalida(1);
    }
    return p;
}

void InicializarArchivosTMP(void) {
    int i;
    tmpFileType *fileInfo;

    /* inicializar archivos */
    if (nTmpFiles < 3) nTmpFiles = 3;
    file = asegurarMalloc(nTmpFiles * sizeof(tmpFileType*));
    fileInfo = asegurarMalloc(nTmpFiles * sizeof(tmpFileType));
    for (i = 0; i < nTmpFiles; i++) {
        file[i] = fileInfo + i;
        sprintf(file[i]->name, FNAME, i);
        if ((file[i]->fp = fopen(file[i]->name, "w+b")) == NULL) {
            perror("io2");
            LimpiarSalida(1);
        }
    }
}

recType *readRec(void) {

    typedef struct iNodeTag {   /* nodo interno */
        struct iNodeTag *parent;/* padre de nodo original */
        struct eNodeTag *loser; /* pérdida externa */
    } iNodeType;

    typedef struct eNodeTag {   /* nodo externo */
        struct iNodeTag *parent;/* padre de nodo externo */
        recType rec;            /* registro de entrada */
        int run;                /* ejecutar número */
        bool valid;             /* registro entrante es válido */
    } eNodeType;

    typedef struct nodeTag {
        iNodeType i;            /* nodo interno */
        eNodeType e;            /* nodo externo */
    } nodeType;

    static nodeType *node;      /* arreglo de selección de nodos */
    static eNodeType *win;      /* nuevo ganador */
    static FILE *ifp;           /* archivo de entrada */
    static bool eof;            /* true si fin de archivo */
    static int maxRun;          /* maximo númeor a correr*/
    static int curRun;          /* número a correr actual */
    iNodeType *p;               /* puntero a nodo interno */
    static bool lastKeyValid;   /* true si lastKey es válido */
    static keyType lastKey;     /* ultima clave escrita */

    /* leer siguiente registro utilizando selección por reemplazo */

    /* verificar primera llamada */
    if (node == NULL) {
        int i;
```

```
        if (nNodes < 2) nNodes = 2;
        node = asegurarMalloc(nNodes * sizeof(nodeType));
        for (i = 0; i < nNodes; i++) {
            node[i].i.loser = &node[i].e;
            node[i].i.parent = &node[i/2].i;
            node[i].e.parent = &node[(nNodes + i)/2].i;
            node[i].e.run = 0;
            node[i].e.valid = false;
        }
        win = &node[0].e;
        lastKeyValid = false;

        if ((ifp = fopen(ifName, "rb")) == NULL) {
            printf("error: archivo %s, no puede ser abierto\n",
ifName);
            LimpiarSalida(1);
        }
    }

    while (1) {

        /* reemplazar ganador anterior con registro actual */
        if (!eof) {
            if (fread(&win->rec, sizeof(recType), 1, ifp) == 1) {
                if ((!lastKeyValid || compLT(win->rec.key, lastKey))
                && (++win->run > maxRun))
                    maxRun = win->run;
                win->valid = true;
            } else if (feof(ifp)) {
                fclose(ifp);
                eof = true;
                win->valid = false;
                win->run = maxRun + 1;
            } else {
                perror("io4");
                LimpiarSalida(1);
            }
        } else {
            win->valid = false;
            win->run = maxRun + 1;
        }

        /* ajustar punteros */
        p = win->parent;
        do {
            bool swap;
            swap = false;
            if (p->loser->run < win->run) {
                swap = true;
            } else if (p->loser->run == win->run) {
                if (p->loser->valid && win->valid) {
                    if (compLT(p->loser->rec.key, win->rec.key))
                        swap = true;
                } else {
                    swap = true;
                }
            }
            if (swap) {
```

```
                /* p podría ser el ganador */
                eNodeType *t;

                t = p->loser;
                p->loser = win;
                win = t;
            }
            p = p->parent;
        } while (p != &node[0].i);

        /* fin de corrida? */
        if (win->run != curRun) {
            /* win->run = curRun + 1 */
            if (win->run > maxRun) {
                /* find e salida */
                free(node);
                return NULL;
            }
            curRun = win->run;
        }

        /* salida del tope del árbol */
        if (win->run) {
            lastKey = win->rec.key;
            lastKeyValid = true;
            return &win->rec;
        }
    }
}

void Corrida(void) {
    recType *win;       /* ganador */
    int fileT;          /* último archivo */
    int fileP;          /* siguiente a ultimo archivo */
    int j;              /* selecciona file[j] */


    /* corrida inicial utiliza selección por reemp[lazo
     * corrida es escrita por distribución FIbonacci.
     */

    /* inicializar estructuras */
    fileT = nTmpFiles - 1;
    fileP = fileT - 1;
    for (j = 0; j < fileT; j++) {
        file[j]->fib = 1;
        file[j]->dummy = 1;
    }
    file[fileT]->fib = 0;
    file[fileT]->dummy = 0;

    level = 1;
    j = 0;


    win = readRec();
    while (win) {
        bool anyrun;
```

```
        anyrun = false;
        for (j = 0; win && j <= fileP; j++) {
            bool run;

            run = false;
            if (file[j]->valid) {
                if (!compLT(win->key, file[j]->rec.key)) {
                    /* añadir a corrida existente */
                    run = true;
                } else if (file[j]->dummy) {
                    /* inciar nueva corrida*/
                    file[j]->dummy--;
                    run = true;
                }
            } else {
                /* primera corrida en archivo */
                file[j]->dummy--;
                run = true;
            }

            if (run) {
                anyrun = true;

                /* corrida fallida */
                while(1) {
                    if (fwrite(win, sizeof(recType), 1, file[j]->fp) !=
1) {
                        perror("io3");
                        LimpiarSalida(1);
                    }
                    file[j]->rec.key = win->key;
                    file[j]->valid = true;
                    if ((win = readRec()) == NULL) break;
                    if (compLT(win->key, file[j]->rec.key)) break;
                }
            }
        }

        /* si no puede correr, suba un nivel */
        if (!anyrun) {
            int t;
            level++;
            t = file[0]->fib;
            for (j = 0; j <= fileP; j++) {
                file[j]->dummy = t + file[j+1]->fib - file[j]->fib;
                file[j]->fib = t + file[j+1]->fib;
            }
        }
    }
}

void RebobinarArchivo(int j) {
    /* rebobinar file[j] y leer primer registro */
    file[j]->eor = false;
    file[j]->eof = false;
    rewind(file[j]->fp);
    if (fread(&file[j]->rec, sizeof(recType), 1, file[j]->fp) != 1) {
```

```
        if (feof(file[j]->fp)) {
            file[j]->eor = true;
            file[j]->eof = true;
        } else {
            perror("io5");
            LimpiarSalida(1);
        }
    }
}

void combinarOrdenación(void) {
    int fileT;
    int fileP;
    int j;
    tmpFileType *tfile;

    /* ordenamiento por polyfase */

    fileT = nTmpFiles - 1;
    fileP = fileT - 1;

for (j = 0; j < fileT; j++) {
        RebobinarArchivo(j);
    }

    /* cada pasada através del ciclo significa una corrida */
    while (level) {
        while(1) {
            bool allDummies;
            bool anyRuns;

            /* localizar corrida */
            allDummies = true;
            anyRuns = false;
            for (j = 0; j <= fileP; j++) {
                if (!file[j]->dummy) {
                    allDummies = false;
                    if (!file[j]->eof) anyRuns = true;
                }
            }

            if (anyRuns) {
                int k;
                keyType lastKey;


                while(1) {
  /* cada pasada escribe un regisro a file[fileT] */

                  /* lcoalizar clave más pequeña */
                    k = -1;
                    for (j = 0; j <= fileP; j++) {
                        if (file[j]->eor) continue;
                        if (file[j]->dummy) continue;
                        if (k < 0 ||
                        (k != j && compGT(file[k]->rec.key, file[j]-
>rec.key)))
                            k = j;
```

```
            }
            if (k < 0) break;

            /* escriba registro */
            if (fwrite(&file[k]->rec, sizeof(recType), 1,
                    file[fileT]->fp) != 1) {
                perror("io6");
                LimpiarSalida(1);
            }

            /* reemplaze registro */
            lastKey = file[k]->rec.key;
            if (fread(&file[k]->rec, sizeof(recType), 1,
                    file[k]->fp) == 1) {
                /* verificar fin de corrida */
                if (compLT(file[k]->rec.key, lastKey))
                    file[k]->eor = true;
            } else if (feof(file[k]->fp)) {
                file[k]->eof = true;
                file[k]->eor = true;
            } else {
                perror("io7");
                LimpiarSalida(1);
            }
        }

        /* arreglar dummies */
        for (j = 0; j <= fileP; j++) {
            if (file[j]->dummy) file[j]->dummy--;
            if (!file[j]->eof) file[j]->eor = false;
        }

    } else if (allDummies) {
        for (j = 0; j <= fileP; j++)
            file[j]->dummy--;
        file[fileT]->dummy++;
    }

    /* fin de corrida */
    if (file[fileP]->eof && !file[fileP]->dummy) {
        /* completado un nivelfibonocci */
        level--;
        if (!level) {
            /* we're done, file[fileT] contains data */
            return;
        }

        /* fileP está lleno, reabrir un nuevo */
        fclose(file[fileP]->fp);
        if ((file[fileP]->fp = fopen(file[fileP]->name, "w+b"))
                == NULL) {
            perror("io8");
            LimpiarSalida(1);
        }
        file[fileP]->eof = false;
        file[fileP]->eor = false;

        RebobinarArchivo(fileT);
```

```
                /* f[0],f[1]...,f[fileT] <-- f[fileT],f[0]...,f[T-1] */
                tfile = file[fileT];
                memmove(file + 1, file, fileT * sizeof(tmpFileType*));
                file[0] = tfile;

                /* inciair nueva corrida */
                for (j = 0; j <= fileP; j++)
                    if (!file[j]->eof) file[j]->eor = false;
            }
        }

    }
}


void OrdenaciónExterna(void) {
    InicializarArchivosTMP();
    Corrida();
    combinarOrdenación();
    termTmpFiles(0);
}

int main(int argc, char *argv[]) {

    /* línea de comandos:
     *
     *   ext ifNombre ofNombre nTmps nNodos
     *
     *   ext in.dat out.dat 5 2000
     *       lee in.dat, ordena registros utilizando 5 archivos y 2000
nodos, genera salida en out.dat
     */
    if (argc != 5) {
        printf("%s ifNombre ofNombre nTmps nNodos\n", argv[0]);
        LimpiarSalida(1);
    }

    ifName = argv[1];
    ofName = argv[2];
    nTmpFiles = atoi(argv[3]);
    nNodes = atoi(argv[4]);

    printf("OrdenaciónExterna: nFiles=%d, nNodes=%d, lrecl=%d\n",
        nTmpFiles, nNodes, sizeof(recType));

    OrdenaciónExterna();

    return 0;
}
```

Programa para generar datos de prueba:

```
/* ordenación de datos externa */

#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int i, *rec, nrecs, lrecl;
    char *fileName;
    FILE *fp;

    /* línea de comandos:
     *
     *   extd fileName lrecl nrecs
     *
     *   extd in.dat 8 2000
     *       crea archivo in.dat, con 2000 registros de 8-bytes
     */
    if (argc != 4) {
        printf("%s NombreArchivo lrecl nrecs\n", argv[0]);
        exit(1);
    }
    fileName = argv[1];
    lrecl = atoi(argv[2]);
    nrecs = atoi(argv[3]);
    if ((rec = malloc(lrecl)) == NULL) {
        printf("error: malloc(%d)\n", lrecl);
        exit(1);
    }
    printf("creando %d registros de tamaño %d...", nrecs, lrecl);


    srand(1);
    if ((fp = fopen(fileName, "wb")) == NULL) {
        printf("\nerror: archivo %s, no puede abrirse\n", fileName);
        exit(1);
    }
    for (i = 0; i < nrecs; i++) {
        *rec = rand();
        if (fwrite(rec, lrecl, 1, fp) != 1) {
            printf("\nerror: escribiendo archivo %s\n", fileName);
            exit(1);
        }
    }
    fclose(fp);
    free(rec);
    printf(" Listo\n");
    return 0;
}
```

Programa para chequear la salida:

```
/* chequear ordenación externa */


#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int nrecs, lrecl, i;
    int *key, prevKey;
    char *fileName;
    FILE *fp;

    /* línea de comandos:
     *
     *   extc NombreArchivo lrecl nrecs
     *
     *   extc out.dat 8 2000
     *   chequea orden en archivo out.dat, con 2000 registros de 8-
bytes
     */
    if (argc != 4) {
        printf("%s NOmbreArchivo lrecl nrecs\n", argv[0]);
        exit(1);
    }
    fileName = argv[1];
    lrecl = atoi(argv[2]);
    nrecs = atoi(argv[3]);
    if ((key = malloc(lrecl)) == NULL) {
        printf("error: malloc(%d)\n", lrecl);
        exit(1);
    }
    if ((fp = fopen(fileName, "rb")) == NULL) {
        printf("error: Archivo %s, no puede abrirse\n", fileName);
        exit(1);
    }

    printf("%s: verificando registros, lrecl=%d...", fileName, lrecl);
    prevKey = -1;
    i = 0;
    while (1) {
        if (fread(key, lrecl, 1, fp) != 1) {
            if (feof(fp)) break;
            printf("\nerror: leyendo archivo %s\n", fileName);
            exit(1);
        }
        i++;
        if (*key < prevKey) {
            printf("\nerror: key[%d]=%d, key[%d]=%d\n",
                i-1, prevKey, i, *key, prevKey);
            exit(1);
        }
        prevKey = *key;
    }
    fclose(fp);
```

```
    if (i == nrecs) {
        printf(" Correcto\n");
        return 0;
    } else {
        printf(" Error, nrecs=%d, measured %d\n", nrecs, i);
        return 1;
    }
}
```

4.4.2 Arboles B (B-Trees)

Los diccionarios para los archivos muy largos residen normalmente en almacenamientos secundarios, tales como un disco. El diccionario es implementado con un índice al archivo actual y contiene las claves y direcciones de registros de datos. Para implementar un diccionario hemos utilizado árboles rojo-negro (red-black trees), reemplazando punteros con compensaciones para el inicio del archivo de índices, y utilizando acceso randómico para apuntar los nodos en el árbol. Sin embargo, todas las transiciones en un enlace podrían implicar un acceso a disco, y podrían tener costos prohibitivos. Vueltas a llamar como I/O de bajo nivel a disco por sectores (típicamente 256 bytes). Podríamos acomodar el tamaño del nodo al tamaño del sector, y grupo de algunas claves juntas en cada nodo para minimizar el número de operaciones de I/O. Este es el principio de los árboles B. Para mejores referencias sobre árboles B consulte Knuth, Cormen y Aho.

Teoría

La figura 4.4.3 ilustra un árbol B con 3 claves por nodo. Las claves en los nodos internos son rodeados de punteros, o compensaciones de registros, a claves que son menores que o mayores que, el valor de la clave. Por ejemplo, todas las claves menores que 22 a la izquierda y todas las mayores que 22 a la derecha. Por simplicidad, no he indicado la dirección del registro asociada con cada clave.

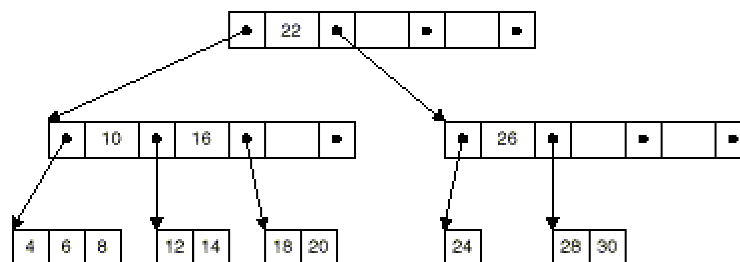


Figura 4.4.3: B-Tree

Nosotros podemos ubicar una clave en este árbol de 2 niveles con tres accesos a disco. Si agrupamos 100 claves/nodos, podemos buscar mas de 1,000,000 claves en solo tres lecturas. Para asegurar este almacenamiento apropiado, debemos mantener un árbol balanceado durante la inserción y eliminación. Durante la inserción, debemos examinar los nodos hijos para verificar que estos están habilitados para almacenar un nodo adicional. Si no, entonces un nuevo nodo hermano es adicionado al árbol, y las claves del hijo son redistribuidas para hacerle sitio al nuevo nodo. Cuando descendamos por inserción y la raíz este llena, entonces la raíz es desparramada al nuevo hijo, y el nivel del árbol es incrementado. Una acción similar es realizada en la eliminación, donde los nodos hijos podrían ser absorbidos por la raíz. Esta técnica para alterar la longitud del árbol mantiene un árbol balanceado.

	B-Tree	**B-*Tree**	**B+Tree**	**B++Tree**
Datos almacenados	algún nodo	algún nodo	solo hoja	solo hoja
en inserción, dividir	1 x 1 -> 2 x 1/2	2 x 1 -> 3 x 2/3	1 x 1 -> 2 x 1/2	3 x 1 -> 4 x 3/4
en eliminación, saltar	2 x 1/2 -> 1 x 1	3 x 2/3 -> 2x 1	2 x 1/2 -> 1 x 1	3 x 1/2 -> 2 x 3/4

Tabla 4.4.1: Implementaciones B-Tree

Algunas variantes de los árboles B son listadas en la Tabla 4.4.1. el B-Tree *estándar* almacena las claves y los datos en ambos internamente y en su hoja. Cuando descendemos el árbol durante la inserción, un nodo hijo lleno es primeramente redistribuido a los nodos adyacentes. Si el nodos adyacente está también lleno, entonces un nuevo nodo es creado, y un ½ de las claves en el hijo son movidas al nuevo nodo creado. Durante la eliminación, los hijos que están medio llenos primero probamos obtener claves de los nodos adyacentes. Si los nodos adyacentes están medio llenos, entonces dos nodos son unidos para formar un nodo lleno. Los B-*Trees son similares, solamente los nodos guardan los 2/3 de su capacidad. Estos resultados en mejor utilización de espacio en el árbol, y descuidan el rendimiento.

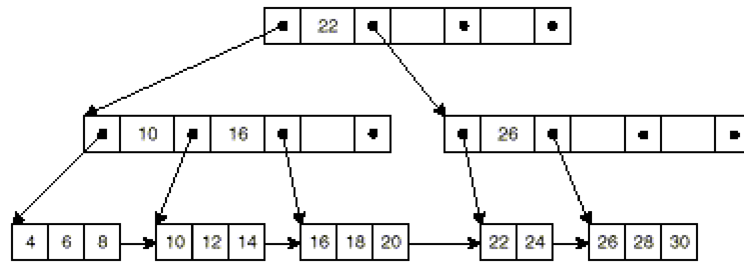


Figura 4.4.4: B-*Tree