UNIVERSIDAD AUTONOMA DE SAN LUIS POTOSI

FACULTAD DE CIENCIAS

Algoritmos y Estructuras de datos

TESIS PROFESIONAL

PARA OBTENER EL TITULO DE:

Ingeniero Electrónico

PRESENTA:

Araceli Arévalo Ramírez

SAN LUIS POTOSI, S. L. P. MARZO DE 2004

UNIVERSIDAD AUTONOMA DE SAN LUIS POTOSI

FACULTAD DE CIENCIAS

Algoritmos y Estructuras de datos

TESIS PROFESIONAL

para obtener el título de

Ingeniero Electrónico

PRESENTA:

Araceli Arévalo Ramírez

SAN LUIS POTOSI, S. L. P. MARZO DE 2004

UNIVERSIDAD AUTONOMA DE SAN LUIS POTOSI

FACULTAD DE CIENCIAS

Algoritmos y Estructuras de datos

TESIS PROFESIONAL

para obtener el título de

Ingeniero Electrónico

PRESENTA:

Araceli Arévalo Ramírez

ASESORES DE TESIS:

Dr. Gerardo Ortega Zarzosa
Fis. Héctor Eduardo Medellín Anaya

SAN LUIS POTOSI, S. L. P. MARZO DE 2004

Dedicatoria

Este trabajo se lo quiero dedicar a mis papás y hermanos con esto les quiero agradecer el que estuvieran conmigo en los momentos más difíciles de mi carrera gracias por hacerme sentir que cuento con su apoyo en todo momento.

Alejandro:
Gracias por estar conmigo en todo momento esto también es para ti por tu invaluable apoyo y tu gran paciencia.

Agradecimientos

A mi familia:
Por su apoyo incondicional en todo momento, en especial a mi mamá porque siempre estuvo conmigo, le agradezco infinitamente la atención que me prestaba al verme trabajar en la realización de este proyecto y durante toda la carrera.

A Alejandro: Por su ayuda y apoyo durante toda la carrera gracias por las veces que tuvimos que desvelarnos para poder terminar este proyecto

Mi sincero agradecimiento al Fis. Héctor Medellín Anaya por la ayuda que me brindo en terminar este trabajo y en la agilización del mismo. Agradezco también al Dr. Gerardo Ortega Zarzosa por su valiosa colaboración.

ÍNDICE

INTRODUCCIÓN

En este trabajo se recopilaron algunos temas selectos de algoritmos y estructuras de datos, con el fin de reforzar los conocimientos adquiridos en cursos como Introducción a la computación o en materias a fin durante la carrera de Ingeniero Electronico, ya que en el transcurso de estos antecedentes no se alcanzan a ver estos temas por falta de tiempo, sin embargo son de gran importancia para todas aquellas personas que esten interesadas en programación avanzada de estructuras de datos.

Iniciamos con los algoritmos de ordenación; el algoritmo del método de ordenación por inserción en listas se describe en un programa en C++ para ordenar una lista de enteros. Se definen clases para nodo y lista y se escriben métodos para insertar nodos en lista, ordenarla y recorrerla.

En el tema de árboles-B se define sus características y operaciones mas comunes, se mencionan las aplicaciones mas importantes el tema incluye descripción grafica de los algoritmos para construir, insertar y eliminar elementos de un árbol-B.

Se incluyen algunas definiciones de conceptos relacionados con grafos y se describe brevemente algunas representaciones de algunos algoritmos comunes, entre ellos el algoritmo Dijkstra que se usa para encontrar el camino mínimo entre dos nodos de un grafo.

Finalmente se ven tres algoritmos Greedy el primero presenta la solución al problema del rompecabezas numérico de 9 casillas y el segundo presenta la solución al problema del juego "Mente Maestra" dicho juego tiene una complejidad NP-completo y el tercero presenta un algoritmo de búsqueda por el método de dispersión; se define el concepto de colisión y se describen algunas técnicas para manejarlas. El algoritmo de comprensión de Huffman es uno de los mas usados en la actualidad, en este trabajo se presenta una descripción del algoritmo y menciono algunas de sus aplicaciones.

Los temas fueron incluidos como material complementario del curso de Introducción a la computación, este trabajo recopila estos temas en un solo documento; se incluye código desarrollado en C++ y pascal para Dos y Windows.

CAPITULO 1

ALGORITMOS DE ORDENACION

CAPITULO 1
ALGORITMOS DE ORDENACION

Introducción

En este capitulo se presentan algunos metodos especiales de ordenación. Se revisa una implementacion del metodo de ordenación por selección. Si bien este metodo es de los más sencillos, la versión que se presenta ordena una lista enlazada. Esto permite que sea aplicado en los casos en que el tamaño de un arreglo este limitado. Se presenta aplicación desarollada en C++.

En los otros dos temas se estudia la construcción y búsqueda en los arboles son un buen ejemplo de una estructura compleja que es relativamente sencillo analizar.

1.1 Método de ordenación por inserción en listas

Para ordenar una lista de enteros utilizando el método de inserción. Se definen clases para nodo y lista y se escriben métodos para insertar nodos en la lista, ordenarla y recorrerla. Se utiliza una lista con un solo enlace.
También se hará uso de un programa en C++ para hacer uso de este método.

La clase nodo quedó definida como se muestra en el listado 1.

```
class nodo{
      int x;
      nodo *sig;
      public:
      void tomadato(int x1){x = x1;};
      void tomanodo(nodo *sig1){sig = sig1;};
      int damedato(){return x;};
      nodo *damenodo(){return sig;};
};
```

Listado 1

Agregamos la definición de los métodos directamente en la declaración. La clase lista se muestra en el listado 2.

```
class lista{
      nodo *cab;
      public:
      lista(){cab = NULL;};
      ~lista();
      void insertar(int x1);
      void recorrer();
```

```
        void ordenar();
};

lista::~lista()
{
        nodo *p,*q;

        p = cab;
        while(p!=NULL){
          q = p->damenodo();
          delete p;
          p = q;
        }
}

void lista::insertar(int x1)
{
        nodo *aux;

        aux = new nodo;
        aux->tomadato(x1);
        aux->tomanodo(cab);
        cab = aux;
}

void lista::recorrer()
{
        nodo *aux;

        aux = cab;
        while(aux != NULL){
             cout << aux->damedato()<<"  ";
             aux = aux->damenodo();
        }
        cout << '\n';
}

void lista::ordenar()
{
        nodo *p,*q,*r,*s,*t;

        r = cab;
        q = r->damenodo();
        while(q != NULL){
             p = cab;              //inicia recorrido
             s = p;
             //busca donde insertar q
```

```
            while(((p->damedato())    <    (q->damedato()))    &&
(p!=q)){
                  s = p;
                  p = p->damenodo();
            }
            if(p!=q){
                  t = (q->damenodo());      // t = next(q)
                  r->tomanodo(t);     // next(r) = next(q)
                  if(s!=p){       //insertar q después de s
                        s->tomanodo(q); // next(s) = q
                        q->tomanodo(p); // next(q) = p
                  }
                  else {                  //inserta en la cabeza
                        q->tomanodo(cab);// next(q) = cab
                        cab = q;
                  }
                  q = t;
            }
            else {                        //no hay inserción
                  r = q;
                  q = q->damenodo();
            }
      }
}
```
Listado 2.

En este caso solo definimos el constructor en la declaración. El destructor elimina todos los nodos de la lista. El método de inserción agrega nodos al inicio de la lista sin hacer otro proceso. Si los nodos llegan desordenados la lista crecerá en desorden. El método de ordenación realiza el siguiente proceso:

- Recorrer la lista con un apuntador empezando en el segundo nodo
- Comparar el nodo actual con todos los anteriores, si es menor, insertarlo antes del último nodo comparado

La figura 1 muestra como ocurre la inserción en la cabeza de la lista. La figura 2 muestra como se inserta un nodo después de otro. Se requirieron cinco apuntadores para llevar a cabo estos procesos. El programa principal construye una lista con números aleatorios y la ordena. Se muestra en el listado 3.

```
void main()
{
      lista list;
      int n;
```

```
    clrscr();
    cout << "Teclee número de elementos:";
    cin >> n;
    for(int i=0; i < n; i++)
         list.insertar(random(2*n));
    list.recorrer();
    list.ordenar();
    list.recorrer();
    getch();
}
```

Listado 3.

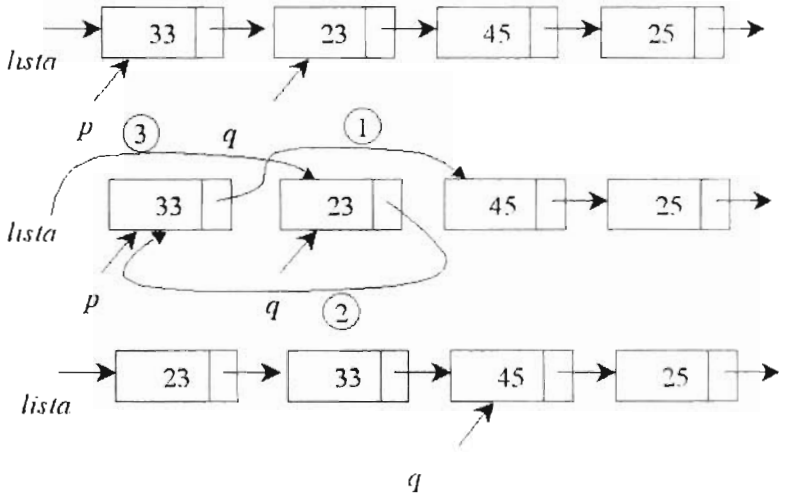


Figura 1. Inserción de un nodo a la cabeza de la lista. Los números encerrados en círculos indican el orden en que se modifican los enlaces.

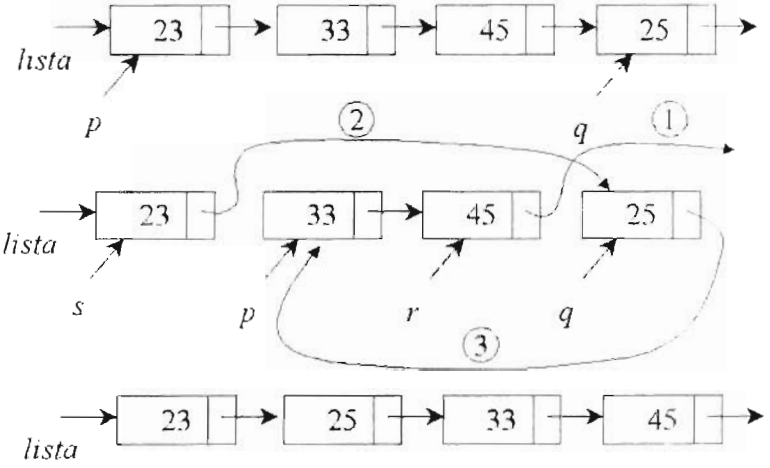


Figura 2. Inserción después del nodo s.

1.2 Árboles-B

Definición de árboles-B

Antes de definir un árbol-B es conveniente definir un árbol de búsqueda de acceso múltiple. En [1] se define un *árbol de búsqueda de acceso múltiple* como un árbol general en el que cada nodo tiene n o menos subárboles y $n-1$ claves. Además, si $s_0, s_1, \ldots, s_{n-1}$ son los subárboles de un nodo y $k_0, k_1, \ldots, k_{n-2}$, son las llaves en orden ascendente, todas las llaves den subárbol s_j son menores k_{j-1} y mayores o iguales a k_j. El subárbol s_j se llama el *subárbol izquierdo* de k_j y su raíz el *hijo izquierdo* de k_j. Similarmente s_j se llama el *subárbol derecho* de k_{j-1} y su raíz el *hijo izquierdo* de k_{j-1}. El orden de un árbol de búsqueda de acceso múltiple es el número máximo de ramas que pueden partir de un nodo.

Un árbol-B es un árbol de búsqueda de acceso múltiple que cumple con lo siguiente[2]:

1. Todos los nodos terminales, (nodos hoja), están en el mismo nivel.
2. Todos los nodos intermedios, excepto la raíz, deben tener entre $n/2$ y n ramas no nulas.
3. El máximo número de claves por nodo es $n-1$.
4. El mínimo número de claves por nodo es $(n/2)-1$.
5. La profundidad (h) es el número máximo de consultas para encontrar una clave.

Búsqueda en un árbol-B

El siguiente algoritmo recursivo de búsqueda es tomado de [1].

> "Cada nodo contiene un solo campo entero, un número variable de campos apuntadores y un número variable de campos llaves. Si *node*(p) es un nodo, el campo entero *numtrees*(p) es igual al número de subárboles de *node*(p). *numtrees*(p) siempre es menor o igual que el orden del árbol, n. Los campos apuntadores *son*(p, 0) hasta *son*(p, *numtrees*(p - 1)) apuntan a los subárboles de *node*(p). Los campos llaves $k(p, 0)$ hasta $k(p,$ *numtrees*(p)—2) son las llaves contenidas en *node*(p) en orden ascendente. El subárbol al que apunta *son*(p, i) (para, i entre 1 y *numtrees*(p) - 2 inclusive) contiene todas las llaves del árbol entre $k(p, i-1)$ y $k(p, i)$. *son*(p, 0) apunta a un subárbol que contiene sólo llaves menores que $k(p, 0)$ y *son*(p, *numtrees*(p) - 1) apunta a un subárbol que contiene sólo llaves mayores que $k(p,$ *numtrees*(p) - 2).
>
> También suponemos una función *nodesearch*(p, *key*) que da como resultado el menor entero j, tal que $key <= k(p, j)$, o *numtrees*(p) - 1 si *key* es mayor que todas las llaves en *node*(p). El siguiente algoritmo recursivo es para una función *search*(*tree*) que da como resultado un apuntador al nodo que contiene *key* (o -1 [representando nulo 0] si no hay tal nodo en el árbol) y hace la variable global *position* igual a la posición *de key* en ese nodo.

```
p = tree;

if (p == nuil) {
     position = -1;
     return(-1);
}    / * fin de if * /
i = aodesearch(p, key);
If (1 < numtrees(p) - 1 && key == k(p,1)) {
     position = 1;
     return(p);
}   / * fin de if * /
return( search ( son(, p, i)));
```

Construcción

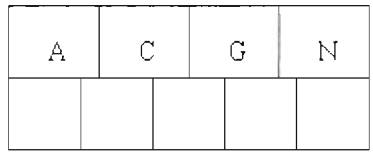
La construcción de un árbol-B crea un nodo vacío para la raíz, el nodo no tiene claves y es un nodo hoja. Solo en este caso se permite que se violen las restricciones del árbol-B.

```
B-Tree-Create(T)
x = NuevoNode();
leaf(x) <- TRUE;
n[x] = 0;
raiz(T) = x
```

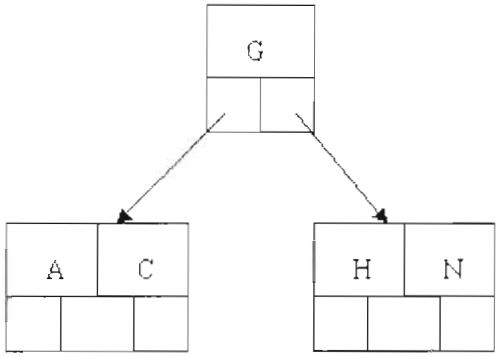
Inserción en árboles-B

Para insertar un nuevo elemento en un árbol-B, primero hay que determinar si el elemento ya ha sido insertado, sino está en el árbol, la búsqueda terminará en una hoja. Si la hoja tiene espacio, simplemente se inserta. Esto requiere que se muevan algunas claves en el nodo. Si no hay espacio para insertar la clave, el nodo debe ser dividido con la mitad de las claves en un nuevo nodo ala derecha de este nodo. La clave del medio es movida al nodo padre. Si en el padre no hay espacio, el proceso se repetirá en él. Note que al agregar un nodo, no solo las llaves son movidas sino también los apuntadores a los hijos. Si se divide el padre, la clave mediana se mueve a la raíz, incrementando la altura en uno.

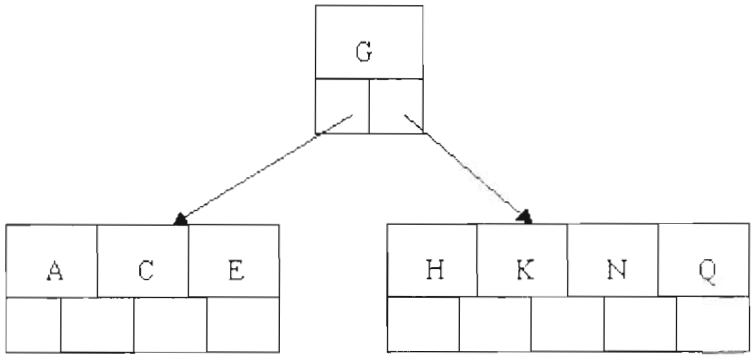
El siguiente ejemplo, muestra este proceso. Se insertarán las claves en un árbol-B de orden 5: C N G A H E K Q M F W L T Z D P R X Y S. Las primeras cuatro se insertan en un nodo:



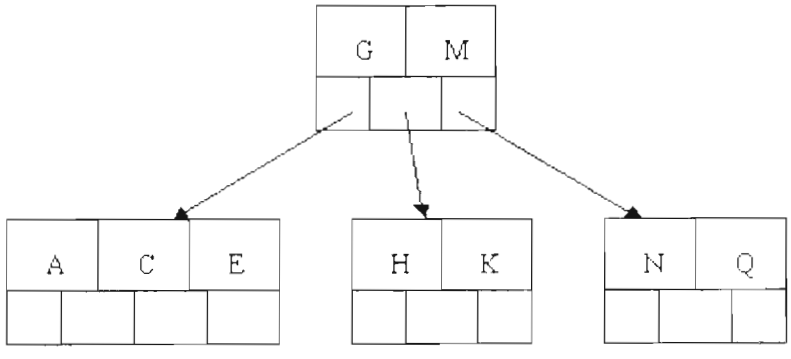
Cuando se quiere insertar H, no hay espacio, se divide el nodo y se inserta G en la raíz.



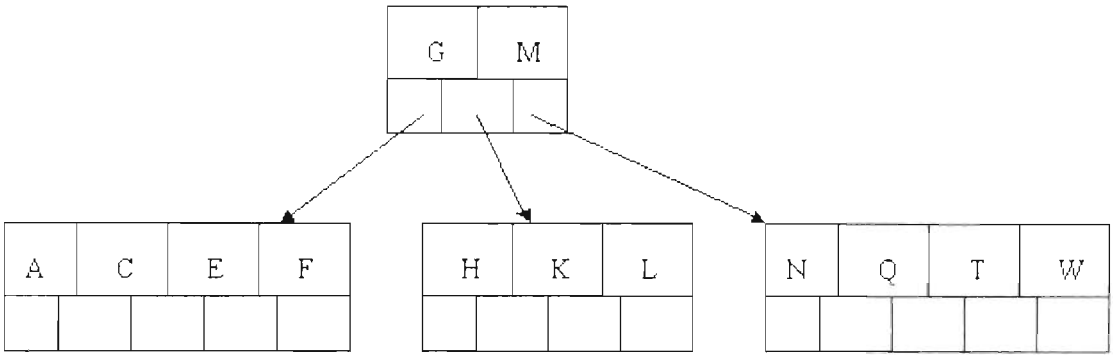
La inserción de E, K y Q no requiere divisiones:



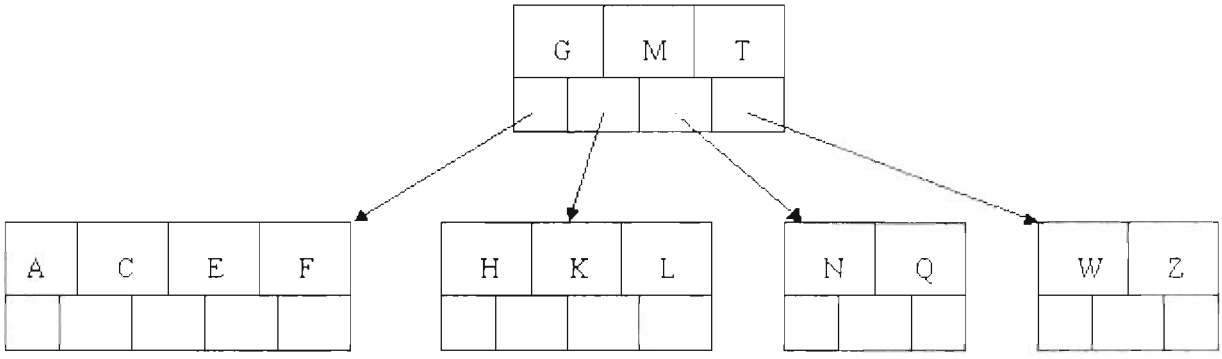
La inserción de M requiere división, y además M es la clave media:



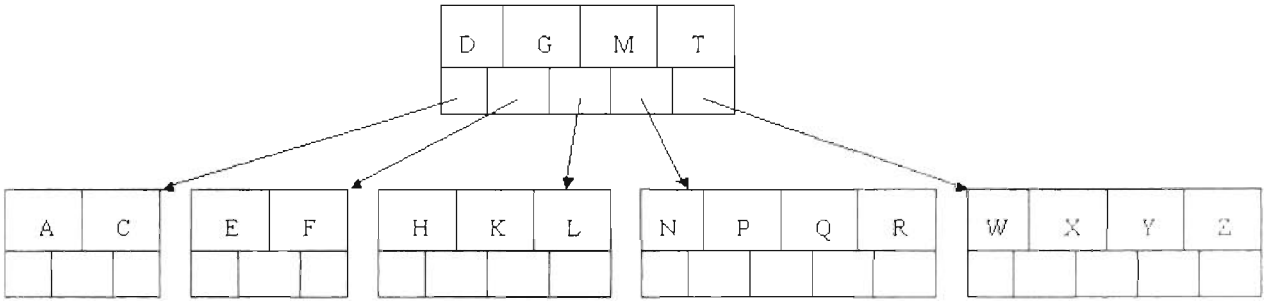
La inserción de F, W, L y T no requiere divisiones:



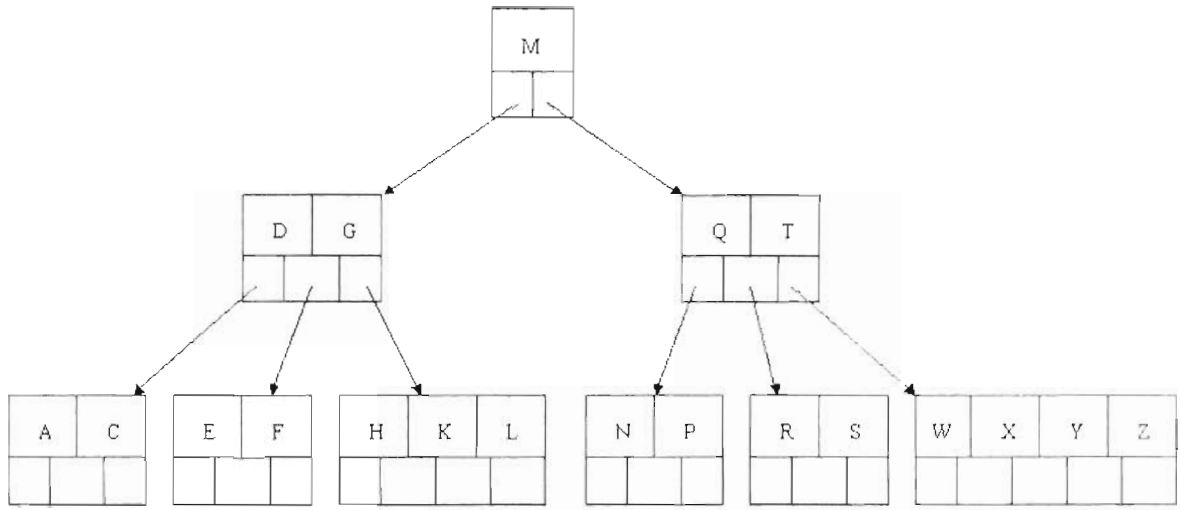
Cuando se agrega Z se debe dividir el nodo de la derecha y T pasa al padre:



Cuando se agrega D se debe dividir el nodo de la izquierda y D pasa al padre:

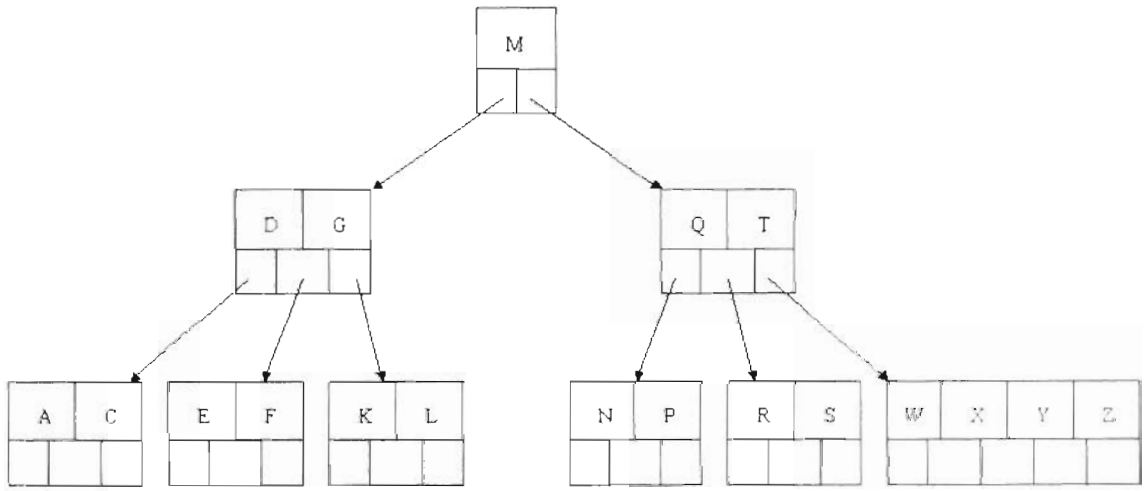


Finalmente, cuando se inserta S, el nodo con N, P, Q y R se divide, mandando la mediana Q al padre. Sin embargo el nodo padre está lleno, debe dividirse, mandando a M a la nueva raíz. Note que los tres punteros del nodo con D y G permanecen sin cambio:

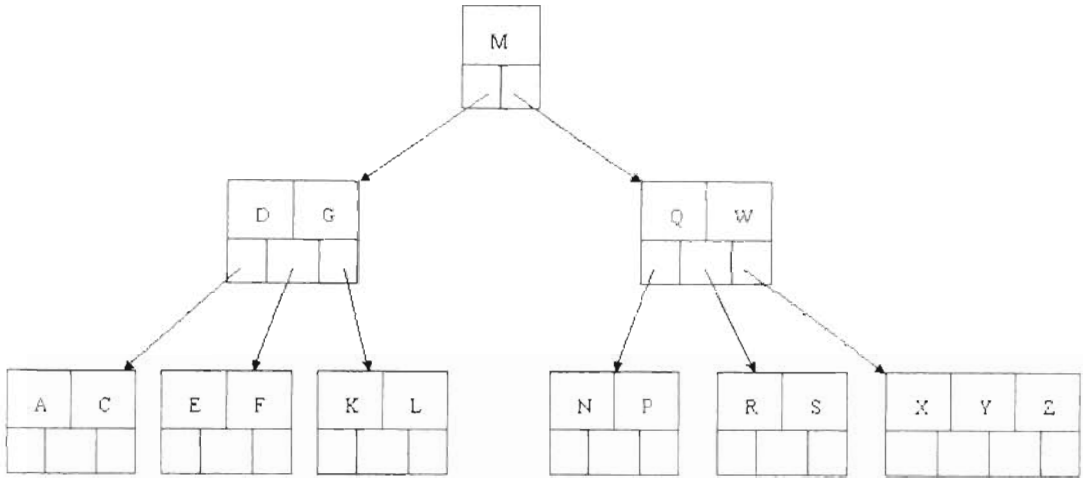


borrado de claves

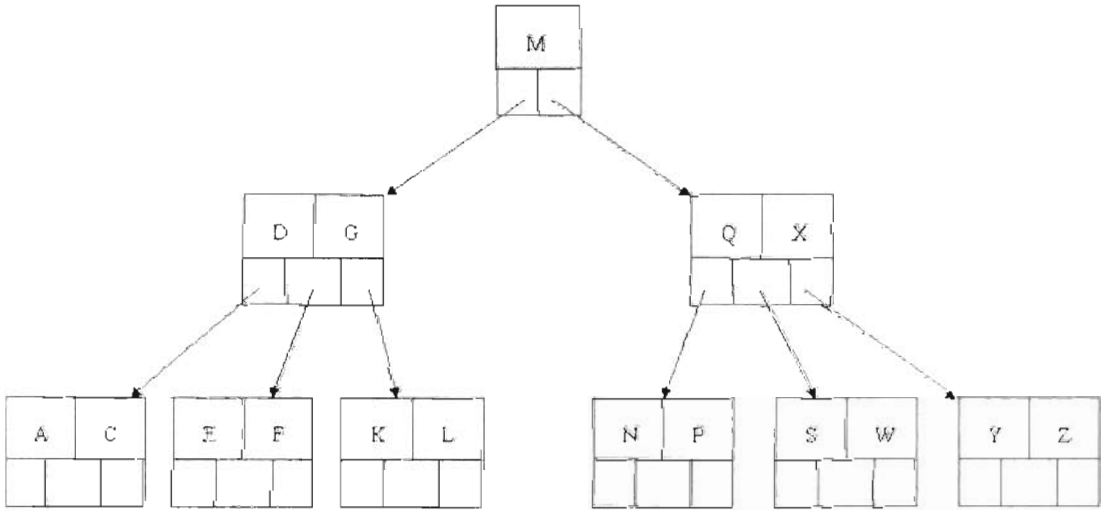
El borrado del nodo H del árbol anterior requiere solo el recorre las llaves K y L un lugar a la izquierda:



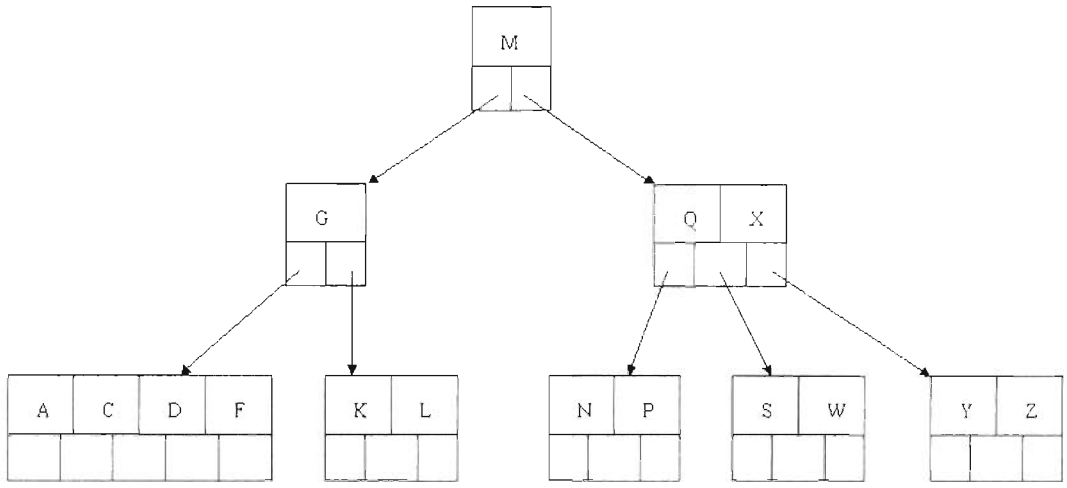
El borrado de T requiere localizar su sucesor (W) y moverlo al lugar de T:



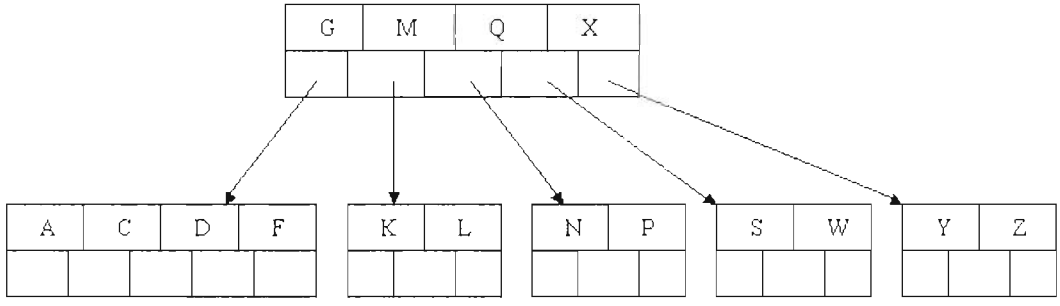
El borrado de R dejaría un nodo con una sola llave, que no es aceptable en un árbol-B de orden 5. Si el hermano a la derecha o izquierda tiene un nodo extra ,podemos borrar una llave del padre y mover una llave desde el hermano. En nuestro caso W se mueve a S y X reemplaza a W en el padre:



Finalmente borremos E. En este caso la hoja resultante debe combinarse con alguno de sus hermanos. Esto incluye mover el padre entre las dos hojas:



Nuevamente esto deja un nodo con una sola llave. Si el hermano de G tuviera tres llaves, se podría subir q a la raíz y bajar M con G. Note que tendrá que pasarse el nodo con N y P como hijo izquierdo de M. En nuestro caso el hermano solo tiene dos llaves, por lo tanto se deberá mover M hacia abajo y juntarlo con el nodo Q X:



Aplicaciones

Los árboles-B tienen aplicaciones en bases de datos. Esta estructura está diseñada para utilizarse en disco. Los árboles-B reales son de orden mucho mayor que 5. Un árbol de orden 100 de dos niveles puede almacenar cerca de 10,000 claves, uno de tres niveles 1,000,000 y uno de 4 niveles 100,000,000 de claves. Si en los nodos se almacenan también los registros, sería difícil mantener todo el árbol en memoria principal.

1.3 Implementación de árboles-B para ordenación

Los árboles-B son ampliamente utilizados como estructuras para optimizar las búsquedas en bases de datos. Generalmente los árboles-B se utilizan en memoria externa. En este trabajo se implemento un árbol-B en memoria principal, solo se utilizó el disco para almacenar los datos de entrada.

El algoritmo utilizado está basado en [1]. Se definió el tipo nodo como un registro con los siguientes campos: *numHijos* para contar el número de llaves en un nodo más 1 (o hijos que potencialmente puede tener el nodo), *padre* un apuntador al padre, *indice* el número de hijo que le corresponde al nodo, *llave* un arreglo de 1 a *ORDEN* -1 con las llaves almacenadas en el nodo, *p* un arreglo de 1 a *ORDEN* con los apuntadores a los hijos del nodo. La declaración es la siguiente:

```
PNodo = ^TNodo;
TNodo = record
  numHijos : 0..ORDEN;
  padre : PNodo;
  indice : integer;
  llave : array [1..ORDEN-1] of TLlave;
  p : array [1..ORDEN] of PNodo;
end;
```

El árbol en sí, es un objeto con un solo miembro dato, *raiz* que es el apuntador a la raíz del árbol. Los métodos se describen en la tabla 1.

Tabla 1.

Método	Descripción
function buscar(arbol: PNodo; Llave: TLlave; var posicion: integer; var Encontrado: boolean):PNodo;	Esta función es la encargada de buscar una llave en el árbol. Regresa el apuntador al nodo donde está la llave y la posición dentro del arreglo de llaves. Si la llave no se encuentra, regresa la última hoja visitada.
function BuscaNodo(p: PNodo:Llave: TLlave): integer;	Esta función localiza la llave más pequeña en un nodo mayor o igual que el argumento de la búsqueda.
function crearArbol(Llave: TLlave): PNodo;	Crea un árbol con un solo nodo y regresa un apuntador al mismo. Inicia los apuntadores a nil y el contador de hijos a 2.
procedure Insertar(llave1: TLlave; s : PNodo; posicion : integer);	Inserta una llave en un árbol-B. Procedimiento de inserción en árboles B.
procedure insertaNodo(nd : PNodo;pos: integer; nuevaLlave: TLlave;nuevoNodo: PNodo);	Inserta una nueva llave dentro de la posición pos de un nodo no lleno. El parámetro nuevoNodo apunta a un subárbol que debe insertarse a la derecha de la nueva llave. Las llaves restantes y los subárboles en las posiciones pos o mayores se recorren una posición.
procedure split(nd : PNodo; pos : integer; nuevaLlave:TLlave;var nuevoNodo,nd2: PNodo; var mediaLlave: TLlave);	Divide en dos a un nodo. Las n/2 llaves menores quedan en el nodo nd, y las demás son colocadas en un nuevo nodo nd2.
Procedure copy(nd1:PNOdo;primera,ultima:integer:nd2:PNOdo);	Copia las llaves del nodo nd1 desde primera hasta última en las llaves de nd2 de la 1 hasta ultima-primera+1. También debe actualizar el apuntador al padre de cada hijo que se copie y el indice.
procedure recorre(arbol:PNodo);	Procedimiento recursivo para recorrer el árbol.
procedure InsertaLlave(Llave:TLlave);	Este es el procedimiento principal para insertar una llave. Primero busca en el árbol si la llave se encuentra o no, si no se encuentra la inserta.
destructor destruir;	Método destructor. Destruye el árbol liberando la memoria utilizada.

Descripción del algoritmo

La dependencia de los procedimientos es la que se muestra en la figura 1. El método *InsertaLlave* llama a *buscar* para localizar la llave en el árbol, éste llama al el método *BuscaNodo* busca una llave dentro de un nodo. Si la llave se localiza simplemente termina el método de inserción, si no se localiza se llama al método principal de inserción que es *Insertar*. Si el nodo no está lleno, se llama a *insertarNodo*, que inserta una llave en un nodo no-lleno. Si el nodo esta lleno, se llama entonces a *split* para dividir el nodo. El método *split* llama a su vez a *copy* que copia la mitad de las llaves y apuntadores en un nuevo nodo y posteriormente llama a *insertaNodo* que inserta la llave del medio en el nodo adecuado. Posteriormente Insertar crea una nueva raíz si es necesario.

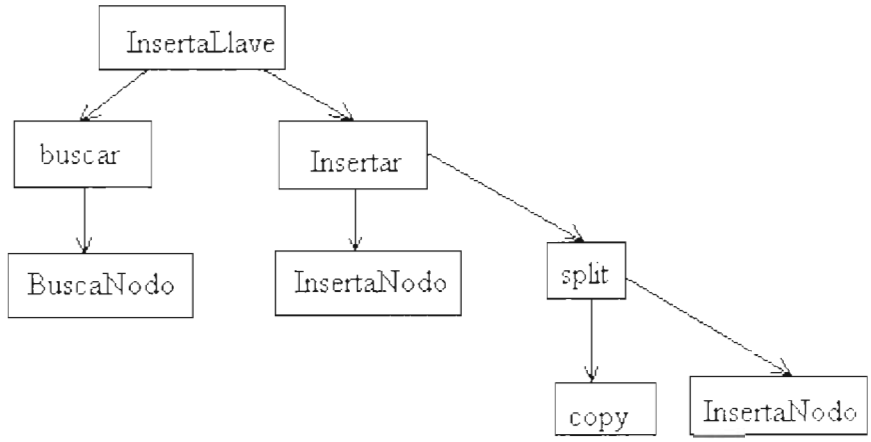


Figura 1. Grafo de dependencia del algoritmo de árboles-B.

El método de recorrido se explica por si mismo, lo mismo sucede con el destructor.

El árbol utiliza el campo *padre* e *indice*, en [1] se recomienda no utilizarlos en el caso de árboles-B definidos en disco, ya que se tendrían que hacer muchos accesos al disco bajando mucho el rendimiento. El siguiente listado muestra la definición del objeto TArbolB.

```
TArbolB = object
private
  raiz:PNodo;
public
  function buscar(arbol:PNodo;Llave:TLlave;
            var posicion:integer;var Encontrado:boolean):PNodo;
  function BuscaNodo(p:PNodo;Llave:TLlave):integer;
  function crearArbol(Llave:TLlave):PNodo;
  procedure Insertar(llave1: TLlave; s : PNodo; posicion : integer);
  procedure split(nd : PNodo; pos : integer;
            nuevaLlave:TLlave;var  nuevoNodo,nd2:PNodo;
            var mediaLlave:TLlave);
  procedure insertaNodo(nd : PNodo;pos:integer;
            nuevaLlave:TLlave;nuevoNodo:PNodo);
```

```
  procedure copy(nd1:PNOdo;primera.ultima:integer;nd2:PNOdo);
  procedure InsertaLlave(Llave:TLlave);
  procedure recorre(arbol:PNodo);
  destructor destruir;
end;
```

Análisis del algoritmo

El árbol-B es balanceado, y todos los nodos contiene $m/2$ llaves (donde m es el orden del árbol), aproximadamente. La tabla 2 [1] muestra el número mínimo y máximo de nodos y llaves que puede contener un árbol-B, $q = (m-1)/2$ y d es el número de niveles.

	Mínimo		Máximo	
Nivel	Nodos	llaves	nodos	llaves
0	1	1	1	$m-1$
1	2	$2q$	m	$(m-1)$m
2	$2(q+1)$	$2q(q+1)$	m^2	$(m-1)\,m^2$
i	$2(q+1)^{i-1}$	$2q(q+1)^{i-1}$	m^i	$(m-1)\,m^i$
total	$1+2((q+1)^d-1)/q$	$2(q+1)^d$	$(m^{d+1}-1)/(m-1)$	$m^{d+1}-1$

La inserción requiere de buscar en todos los niveles, en cada nivel se analiza un solo nodo. En el caso del mínimo de llaves por nodo, si se tienen n llaves, entonces $n = 2(q+1)^d$, de aquí que el número total de accesos sea $d = \log_{q+1}(n/2)$. En cada acceso a un nodo se recorrerá en promedio la mitad de las llaves, esto puede mejorar si se usa búsqueda binaria. En el caso del máximo número de llaves por nodo, es fácil mostrar que el número de accesos es $\log_m(n+1)$. En ambos casos se puede ver que el algoritmo tiene un comportamiento del orden $O(\log n)$.

Resultados

Se incluyeron algunas variables para llevar la contabilidad. Se contaron el número de llaves, nodos, niveles del árbol, cantidad de memoria utilizada y la memoria para almacenar datos. Los resultados para el archivo procesado se muestran a continuación:

Número de llaves: 100
Número de nodos: 38
Número de niveles: 3
Tamaño del nodo: 71 bytes
Memoria utilizada total: 2698 bytes
Memoria utilizada en datos: 1672 bytes
Llaves por nodo: 2.6316

CAPITULO 2

ALGORITMOS DE BUSQUEDA

CAPITULO 2
ALGORITMOS DE BUSQUEDA

Introducción

La búsqueda de información es una de las aplicaciones más comunes en computación. Existen varias formas de buscar información tales como listas enlazadas o árboles. En este capitulo veremos específicamente la tecnica de transformación de claves, llamada tambien Hashing.

2.1 Búsqueda por el método de dispersión de claves (Hashing)

La tecnica de Hashing, se basa en la asignación de un elemento de un arreglo a cada clave de los datos mediante una aplicación H [3,4]. Si C es el conjunto de las claves y D el conjunto de las direcciones, entonces

$$H: C \rightarrow D$$

Una función de dispersión puede generar valores iguales para varias claves. Esto implica que varias claves sean mapeadas a un solo elemento del arreglo. A esta situación se le llama *colisión*, se debe tener un proceso que asigne direcciones alternativas, a esto se le llama *manejo de colisiones*. El *hashing perfecto* es aquel que no produce ninguna colisión, este es posible si se conocen todas las claves de antemano. Este tipo de hashing se aplica mucho en compiladores [5].

Lo deseable es que la función H distribuya los valores generados uniformemente. Una elección es la función módulo. Supóngase que existe una función $ord(k)$ que produce el número de orden de la clave k, si el arreglo tiene índice i entre $0..N\text{-}1$, donde N es la dimensión del arreglo. La función H podría ser:

$$H(k) = ord(k) \textbf{ mod } N$$

Se sugiere en [3] que N sea un número primo para obtener una buena distribución.

Manejo de colisiones. Si dos claves tienen el mismo índice, se dice que existe una colisión. Una solución es tener una lista de elementos que tiene la misma clave, a este método se le llama *encadenamiento directo*. Otro método es buscar nuevos lugares en la tabla, este método se llama *direccionamiento abierto*. Un algoritmo [3] sería el siguiente:

$j := H(k)$; $i := 0$;

```
repetir
        si T[j].clave = k entonces
                encontrado el elemento
        sino
                si T[j].clave = vacío entonces
                        el elemento no está en la tabla
                sino
                        i := i+1;
                        j := H(k) + G(i)
until encontrado o tabla llena
```

Podemos hacer $G(i) = i$, en este caso

$$h0 = H(k)$$
$$h0 = (h0 + i) \textbf{ mod } N$$

Este método se llama *inspección lineal*. Este método tiene la desventaja de que las claves tienden a agruparse alrededor de las claves primarias. Una modificación es utilizar una función cuadrática, el método se llama *inspección cuadrática*

$$h0 = H(k)$$
$$h0 = (h0 + i^2) \textbf{ mod } N$$

En [2] hay un simulador de búsqueda por hash que incluye las dos formas mencionadas. El simulador muestra en número de veces que se llama a la función hash, el número de claves contra el número de "hashes", el número de colisiones durante las inserciones, la distribución de la claves y la tabla de hash.

Comportamiento del método de transformación de claves. En [1] se encuentra que el número de intentos para recuperar (o insertar) una clave es

$$E = -(\ln(1 - \alpha))/\alpha$$

Donde α es el cociente del número de lugares en la tabla que están ocupados entre el total de lugares, este número se llama *factor de carga*. Una forma de eliminar el agrupamiento primario [1] es permitir que la función de redispersión dependa del número de veces que se aplica a un valor. Con *rh*(*i*, *key*) = (*i* + *hkey*) % *tablesize* se elimina el agrupamiento primario pero se produce lo que se conoce como agrupamiento secundario, para eliminarlo se puede utilizar una doble función de hash *hash2*. La siguiente tabla resuma algunos resultado de *E* para estas funciones para búsqueda exitosa e infructuosa[1].

	Exitosa			Infructuosa		
α	***Lineal***	***i + hkey***	***Doble***	***Lineal***	***i + hkey***	***Doble***
25%	1.17	1.16	1.15	1.39	1.37	1.33
50%	1.50	1.44	1.39	2.50	2.19	2.00
75%	2.50	2.01	1.85	7.50	4.64	4.00
90%	5.50	2.85	2.56	50.50	11.40	10.00
95%	10.50	3.52	3.15	200.50	22.04	20.00

Se puede mostrar que la dispersión lineal tiene un comportamiento proporcional a *sqrt*(*tablesize*) para búsqueda exitosa y a (*tablesize*+1)/2 para búsqueda infructuosa. En el segundo caso es proporcional a *ln*(*tablesize*) para búsqueda exitosa y proporcional a 1/(1 – α) para búsqueda infructuosa, y el último caso es proporcional a *ln*(*tablesize*) y (*tablesize*+1)/2 para búsqueda infructuosa

Como se mencionó antes, otro método de manejo de colisiones es mediante una lista enlazada en cada entrada de la tabla de dispersión [1]. Uno de los métodos es el de *dispersión con encadenamiento separado*. Este se usa cuando el número de registros a buscar crece más allá del tamaño de la tabla. El siguiente es el algoritmo en C.

```
Search(KEYTYPE key, RECTYPE rec)
{
     struct nodetype *p, *q, *s;
     i = h(key);
     p = bucket[i];
     while (p != NUL && p->k != key){
          q = p;
          p = p->next;
     }
     if(p->k == key)
          return(p);
     s = getnode();
     s->k = key;
     s->r = rec;
     s->next = NULL;
     if(q == NULL)
          bucket[I] = s;
     else
     q->next = s;
     return(s);
}
```

La figura 1. .muestra el encadenamiento separado. la tabla tiene diez entradas y las claves se insertan en el orden 75, 66, 42, 192, 91, 40, 49, 87, 67, 16, 417, 130, 372, 227.

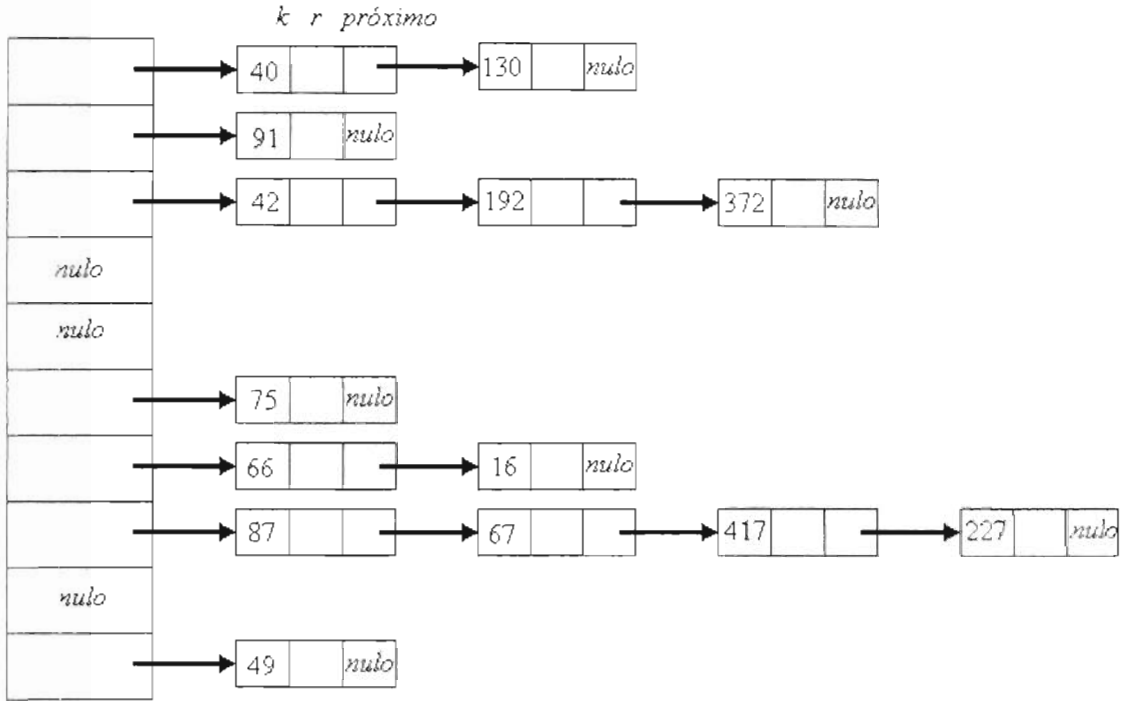


Figura 1.

En [1] se establece que el número promedio de pruebas que se requiere para localizar un elemento existente es aproximadamente $1 + \alpha/2$. No es fácil eliminar elementos de la tabla de dispersión que use redispersion, ya que al eliminar un elemento de la tabla, el lugar que ocupaba quedará disponible para insertar otro, pero se perderá en encadenamiento con las demás claves insertadas.

CAPITULO 3

GRAFOS

CAPITULO 3
GRAFOS

Introducción

En este captulo se presentan algunas definiciones de conceptos relacionados con grafos, se describen brevemente algunas representaciones y algoritmos comunes. Menciono algunas de las aplicaciones de estas estructuras de datos. Tambien se incluye el algoritmo de Dijkstra para encontrarel camino minimo entre dos nodos de ungrafo.

3.1 Grafos

Definiciones

Los siguientes conceptos fueron tomados de [6.7]. Un grafo $G = (V, E)$ es un conjunto finito V de nodos llamados generalmente vértices más un conjunto finito E de aristas, llamadas también arcos. El conjunto de aristas esta formado por pares de vértices. Si los pares son ordenados, se dice que el grafo es un **grafo dirigido** o **digrafo**. La figura 1 muestra un ejemplo de grafo dirigido. Note que una arista puede comenzar y terminar en un mismo vértice.

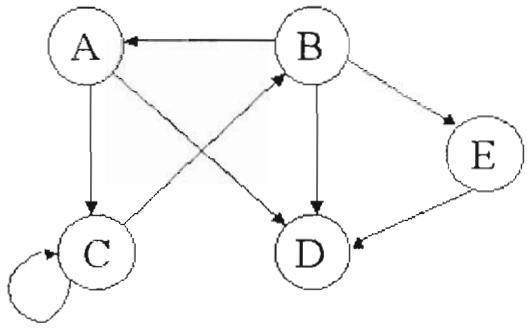


Figura 1. Un grafo dirigido.

Un **grafo no dirigido** es un grafo en que E consiste de pares desordenados. Un grafo no dirigido no contiene autociclos (orejas).

Si (u, v) es una arista de $G = (V, E)$, decimos que v es **adyacente** a u. El grado de un vértice es el número de aristas incidentes a él. Si el grado es cero el vértice se llama **aislado**. En un grafo dirigido el **grado de entrada** es el número de aristas que llegan a él y el **grado de salida** es el número de aristas que salen de él.

Un **camino** de **longitud** k desde el vértice u al u' en un grafo $G = (V, E)$ es una secuencia $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ de vértices tales que $u = v_0$ y $u' = v_k$ y $(v_{i-1}, v_i) \in E$ para $i = 1, 2, \ldots, k$. La longitud de un camino es el número de aristas en el camino. Si hay un camino de u a u', decimos que u' es **alcanzable** desde u. Un camino es **simple** si todos sus vértices son distintos. Si $v_0 = v_k$, el camino es un **ciclo**. Un ciclo es **simple** si todos sus vértices son distintos.

Un grafo dirigido es simple si no contiene autociclos. Un grafo sin ciclos es un grafo **acíclico**.

Un grafo es **fuertemente conectado** si cada pareja de vértices es alcanzable de uno al otro.

Dos grafos $G = (V, E)$ y $G = (V', E')$ son **isomórficos** si existe una biyección $f: V \rightarrow V'$ tal que $(u, v) \in E$ si y solo si $(f(u), f(v)) \in E'$.

Un **grafo completo** es un grafo no dirigido en el cual cada par de vértices es adyacente. Un grafo bipartito es un grafo $G = (V, E)$ no dirigido en el cual V puede ser particionado en dos conjuntos V_1 y V_2 tal que $(u, v) \in E$ implica que o bien $u \in V_1$ y $v \in V_2$ o $v \in V_1$ y $u \in V_2$. Un **hipergrafo** es como un grafo no dirigido. Pero cada **hiperarista**, mas que estar conectando dos vértices, conecta dos subconjuntos arbitrarios de vértices.

Representación

Un grafo puede representarse mediante un arreglo de dos dimensiones[1] (**representación secuancial**). La declaración en C es la siguiente

```
#define MAXNODES 50

struct node{
// información sobre el nodo
};
struct arc{
  int adj;
// información asociada a cada arco
};
struct graph{
  struct node[MAXNODES];
  struct arc arcs[MAXNODES, MAXNODES];
};
struct graph g;
```

El arreglo bidimensional g.arc[][].adj es llamado matriz de adyacencia. Para agregar un arco se hace el elemento arc[][].adj igual a TRUE, y para eliminarlo se hace igual a

FALSE. Mediante operaciones con la matriz de adyacencia pueden determinarse algunas de las características de grafos como la cerradura transitiva.

En los grafos con pesos, llamados red, se desea conocer el camino más corto entre dos nodos *s* y *t*. Se han desarrollado algoritmos para llevar a cabo este cómputo. Una aplicación de grafos con pesos es la determinación del flujo a través de una tubería de agua, en donde elpeso representa la capacidad de la tubería. El algoritmo de Ford-Fulkerson resuelve este problema.

Otra posible representación de grafos es la **representación enlazada**, en la que cada nodo tiene una lista de apuntadores a sus vecinos. La siguiente declaración en C muestra como representar grafos.

```
struct nodetype{
  int info;
  struct nodetype *point;
  struct nodetype *next;
};
struct nodetype nodeptr;
```

3.2 Algoritmo de Dijkstra para encontrar el camino mínimo en grafos

Descripción del programa

El programa fue escrito en Delphi. Se definieron tres unidades. La unidad Unit1 contiene la definición del objeto grafo, este objeto contiene dos miembros datos, uno el número de nodos y una matriz de adyacencia con los valores de los pesos de cada arista. La declaración es la siguiente. También se incluye la declaración del arreglo para las distancias y el conjunto utilizado para el algoritmo de Dijkstra.

```
type
  Nodo = 0..MaxNodos;
  TGrafo = object
  private
    Coste : array[1..MaxNodos,1..MaxNodos]of integer;
    NumNodos : integer;
  public
    procedure PonCoste(i,j:Nodo;c:integer);
    function DameCoste(i,j:Nodo):integer;
    procedure PonNumNodos(n:Nodo);
    function DameNumNodos:Nodo;
    constructor init;
  end;
  TipoDist = array[1..MaxNodos]of integer;
  conjuntoNodos = set of Nodo;
```

El método Dijkstra implementa el algoritmo de Dijkstra, este método, junto con el método de dibujo, la función que regresa el mínimo de dos valores y el de salida del hilo pertenecen a la forma. El resto del código de esta unidad consiste en los eventos de los botones y del menú, estos se explican por sí mismos.

La unidad Unit2 contiene la forma para llevar a cabo la simulación y no contiene código ejecutable. La unidad Unit3 contiene la declaración del hilo de simulación, del que se hablará más adelante, éste debe declararse en una unidad aparte. Se anexan al final el listado de las tres unidades.

El programa cuenta con un menú principal, es este menú hay una opción para cargar un archivo de datos que contenga la descripción de un grafo de hasta 16 nodos. Los archivos deben ser archivos de texto con la descripción de una arista en cada renglón con el formato:

Nodo_inicial nodo_final coste

Los valores del nodo inicial y final deben ser números de 1 a 16. No se realiza chequeo de errores de los datos de entrada. Las otras opciones del menú son salir del programa o desplegar información sobre el programa. El programa muestra la ventana de la figura 1. La ventana muestra en un memo la descripción del grafo y en otro memo el resultado de la ejecución del algoritmo de Dijkstra. Los dos memos son de solo lectura. El usuario puede elegir los nodos extremos del camino a buscar.

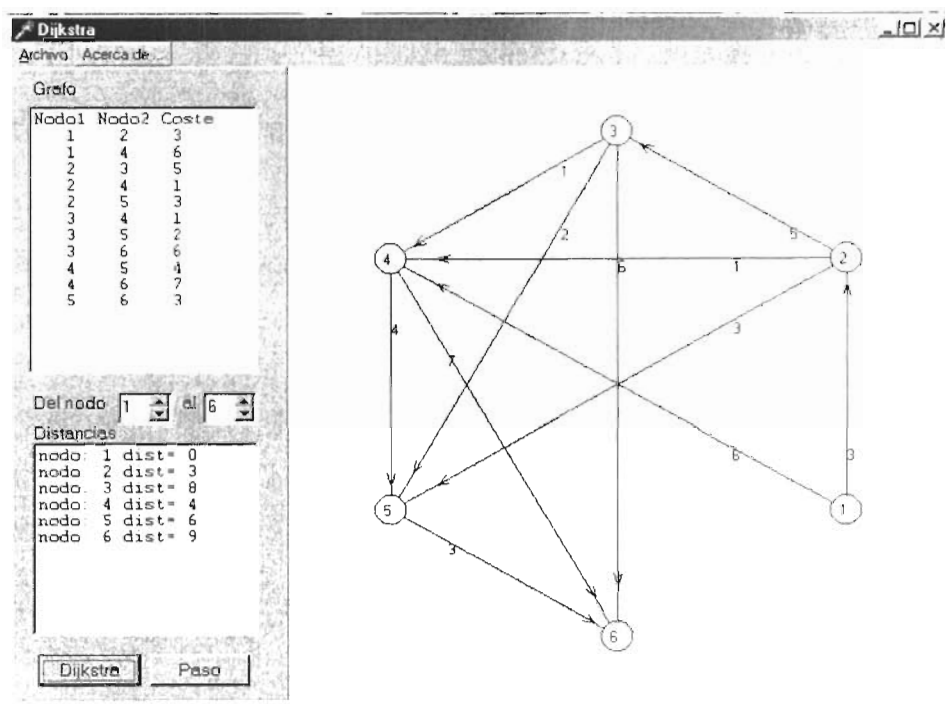


Figura 1. Ventana del programa.

En la parte inferior se encuentran dos botones. El etiquetado "Dijkstra" llama al método que ejecuta el algoritmo de Dijkstra. Al ejecutar el algoritmo el memo inferior muestra el resultado. El algoritmo fue tomado de [6] con una ligera modificación. El lazo repeat del final de algoritmo puede generar un ciclo infinito si no se puede llegar al nodo final. Para evitar esta situación se verifica que se agregue un nodo al conjunto de nodos S en cada paso por el ciclo, si no es así, se informa al usuario que no es posible ir del nodo comienzo al nodo final y se aborta el procedimiento.

El segundo botón etiquetado "Paso" ejecuta el mismo algoritmo paso a paso. Cuando se ejecuta paso a paso, se abre una ventana que despliega el paso del algoritmo que se ha ejecutado y los valores de las variables a cada paso. La implantación de la parte de simulación requirió la creación de un hilo de programa adicional. El botón "Siguiente" de la ventana de simulación despierta al hilo para que se ejecute otro paso del algoritmo. El botón "Terminar" solicita la finalización del hilo y provoca que el algoritmo principal del hilo (Execute) se termine en ese momento.

El programa dibuja el grafo cargado desde disco en la parte derecha de la ventana. El procedimiento de dibujo es bastante simple pero efectivo. Los nodos se distribuyen en un círculo alrededor del centro del área de dibujo, de esta manera se evita que la mayoría de los arcos queden empalmados. El peso de un arco se dibuja cercano al nodo origen, por ejemplo, si se tiene el arco con coste[2,4] = 5 y otro arco con coste[4,2] = 6, se dibujara un '5' más cercano al nodo 2 que al 4 y un '6' más cercano al nodo 4 que al nodo 2.

CAPITULO 4

SOLUCION DE ALGORITMOS VORACES

CAPITULO 4

SOLUCION DE ALGORITMOS VORACES

Introducción

En este capitulo se presenta una solución para un rompecabezas formado por una matriz de tres por tres. La solución se escribió como un programa orientado a objetos en Delphi. La solución se basa en la forma en que el problema se resuelve manualmente. La estructura de datos para representar el rompecabezas es una matriz de tres por tres.

Tambien incluye Solución Greedy al juego de "Mente Maestra" dicho juego tiene una complejida NP-completo. Por ultimo se presenta el algoritmo de comprensión de Huffman es uno de las mas usados en la actualidad aquí se presenta una descripción del algoritmo y menciono algunas de sus aplicaciones.

4.1 Solución al problema del rompecabezas numérico de 9 casillas

El juego consta de un tablero que soporta cuadros con números del 1 al 8. Cada cuadro puede desplazarse libremente hacia la posición vacía. El tablero del rompecabezas se muestra en la figura 1.

5	1	2
6	4	3
8	7	

Figura 1. Rompecabezas numérico.

El objetivo del juego es ordenar el tablero de 8 números que queden como en la figura 2.

1	2	3
4	5	6
7	8	

Figura 2. Posición final del tablero.

El juego introducido por Sam Loyd en 1878 consistiendo de 15 cuadros numerados. Para algunas posiciones no es posible llegar a la solución. Definimos una inversión de orden n como la situación en la que un cuadro con un número i está a la izquierda o arriba de n cuadros con número menor que i y la denotamos por n_i. Definimos el número de permutaciones como:

$$N \equiv \sum_{i=1}^{15} n_i = \sum_{i=2}^{15} n_i$$

Johnson (1879) probó que las permutaciones impares del juego son imposibles, Story (1879) probó que todas las permutaciones pares son posibles. Archer (1999) presentó una prueba simple. "Invertir el orden en un juego de 3x3 se puede probar que toma por lo menos 26 movimientos, la mejor solución requiere 30 movimientos (Gardner 1984, pp. 200 y 206-207). El número de soluciones distintas en 28, 30, 32, ... movimientos son 0, 10, 112, 512." [1].

Análisis de la solución

La solución que propongo se basa en que el problema se puede dividir en etapas. La primera etapa consiste en ordenar el primer renglón, esto es, ordenar los números 1,2 y 3. La segunda etapa consiste en llegar a la posición mostrada en la figura 3, es decir ordenar los números 4 y 5 sin modificar el orden del primer renglón. La posición de los demás número no es importante en esta etapa.

1	2	3
5		8
4	6	7

Figura 3. Ordenación del 4 y 5.

Cabe hacer notar que una vez ordenados los números del 1 al 5 como se muestra en la figura 3 los otros tres solo pueden quedar en las posiciones que se muestran en la figura 4.

1	2	3
5		8
4	6	7

1	2	3
5		6
4	7	8

1	2	3
5		7
4	8	6

Figura 4. Posiciones alcanzables al ordenar los números del 1 al 5.

Por último la cuarta etapa es llevar el tablero a la posición final como se muestra en la figura 2.

Algoritmo

El algoritmo consta de los siguientes pasos:

1. Colocar el espacio vacío en el centro.
2. Ordenar el primer renglón dejando el espacio en el centro
3. colocar el 5 en la posición (2,1)
4. Colocar el 4 en la posición (3,1)
5. Girar 6, 7 y 8 a la posición correcta
6. Colocar 4 y 5 en la posición final
7. Recorrer 7 y 8 a la izquierda.

Implementación

Implemente la solución en Delphi. El programa consta de una ventana con tres botones, un botón para resolver el rompecabezas, otro para generar una posición aleatoria y otro para simular la solución encontrada. Al ejecutar el programa el tablero se encuentra en la posición de la figura 1. El usuario puede llevara cabo los movimientos haciendo clic sobre el número que desea desplazar.

Para la solución definí una clase (objeto en Delphi) `TPuzzle` que contiene todos los atributos y métodos necesarios. El listado 1 muestra la definición.

Listado 1

```
TPuzzle = object
private
  solucion:string[200];
  movimientos:integer;
  tablero,tableroinicial:TTablero;
  hecho:boolean;
public
  procedure init;
  procedure dibuja(canvas:TCanvas);
  procedure cambia(var a,b:integer);
  procedure arriba;
  procedure abajo;
  procedure derecha;
  procedure izquierda;
  procedure resolver;
  procedure revolver;
  procedure secuencia(s:TSecuencia);
  function resuelto:boolean;
end;
```

La clase `TPuzzle` contiene cinco atributos, uno para contar el número de movimientos y otro para el tablero. El tablero se representa como una matriz de tres por tres. Para la simulación definí los otos dos atributos, `tableroInicial` para almacenar el tablero que se va a ordenar y `solucion` que es una cadena que almacena todos los movimientos codificados como se indica más adelante. El atributo `hecho` es verdadero si el procesos de solución ya llego a la posición requerida, este atributo es establecido cada vez que se llama al método `secuencia`. El método `init` inicializa el tablero a la posición inicial. El método `dibuja`, dibuja en un lienzo el tablero. Definí cuatro métodos para hacer los movimientos. Estos métodos mueven el espacio vacío en el sentido del nombre del método, así, `arriba` mueve el espacio vacío hacia arriba, y así sucesivamente. El listado 2 muestra el método `arriba`. El método `cambia`, intercambia dos elementos del tablero.

Listado 2

```
procedure TPuzzle.arriba;
var i,j:integer;
begin
  for i:=2 to 3 do
   for j:=1 to 3 do
     if tablero[i,j]=0 then begin
       cambia(tablero[i,j],tablero[i-1,j]);
       exit;
     end;
end;
```

Existen posiciones del tablero que no son alcanzables desde la posición ordenada y viceversa. Por ejemplo, no es posible partir de la posición de la figura 5(a) a la 5(b) y viceversa. El método `desordenar` revuelve el tablero, para esto realiza 1000 movimientos aleatorios del espacio vacío para garantizar que esta posición es alcanzable desde una posición desordenada.

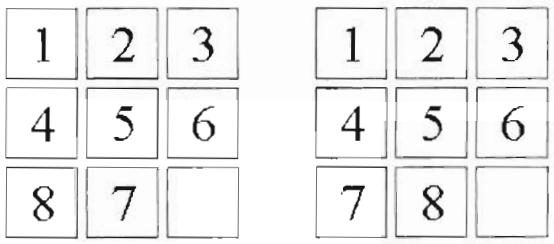


Figura 5. Posiciones mutuamente inalcanzables.

El método `secuencia` ejecuta una secuencia de movimientos. El parámetro de tipo cadena especifica la secuencia. Cada carácter indica una dirección de movimiento. La letra 'a' se utiliza para mover hacia arriba, la 'b' para mover hacia abajo, la 'i' para la izquierda y la 'd' para la derecha. Este método permitió expresar los movimientos complejos de una manera muy compacta.

Para llevar a cabo las primeras 2 etapas, consideré que era necesario comenzar cada movimiento en una posición tal que se conozca la posición del espacio vacío, de tal manera que los demás movimientos pueden expresarse fácilmente. La posición elegida para el espacio vacío es el centro del tablero.

El método que ordena el tablero es `resolver`. La parte que ordena el primer renglón se muestra en el listado 3. Como puede verse, cada paso se realiza solo si el número a mover está fuera de su posición de destino. El resto del código es muy similar.

Listado 3. Ordena el primer renglón.

```
{colocar vacío en el centro}
if tablero[2,2]<>0 then begin
  sec:='';
  if tablero[1,1]=0 then sec:='db'
  else if tablero[1,3]=0 then sec:='ib'
  else if tablero[3,1]=0 then sec:='da'
  else if tablero[3,3]=0 then sec:='ia'
  else if tablero[1,2]=0 then sec:='b'
  else if tablero[2,1]=0 then sec:='d'
  else if tablero[2,3]=0 then sec:='i'
  else if tablero[3,2]=0 then sec:='a';
  secuencia(sec);
end;
{colocar 1 en [1,1]}
if tablero[1,1]<>1 then begin
  sec:='';
  if tablero[1,2]=1 then sec:='iadb'
  else if tablero[1,3]=1 then sec:='addbiiadb'
  else if tablero[2,1]=1 then sec:='aibd'
  else if tablero[2,3]=1 then sec:='daibiadb'
  else if tablero[3,1]=1 then sec:='ibdaaibd'
  else if tablero[3,2]=1 then sec:='biadaibd'
  else if tablero[3,3]=1 then sec:='bdaibiadaibd';
  secuencia(sec);
end;
{colocar 2 en [1,2]}
if tablero[1,2]<>2 then begin
  sec:='';
  if tablero[1,3]=2 then sec:='adbi'
  else if tablero[2,1]=2 then sec:='iadbdaiibdadbi'
  else if tablero[2,3]=2 then sec:='daib'
  else if tablero[3,1]=2 then sec:='biadbdaaib'
  else if tablero[3,2]=2 then sec:='bdaaib'
  else if tablero[3,3]=2 then sec:='dbiadaib';
  secuencia(sec);
end;
{colocar 3 en [1,3]}
```

```
if tablero[1,3]<>3 then begin
  sec:='';
  if tablero[2,1]=3 then sec:='iadbdaiibd'
  else if tablero[2,3]=3 then sec:='iadddbiaibd'
  else if tablero[3,1]=3 then sec:='biadbiaadbdaiibd'
  else if tablero[3,2]=3 then sec:='biaadbdaiibd'
  else if tablero[3,3]=3 then sec:='dbiaiaddbiaibd';
  secuencia(sec);
end;
```

Resultados

Tomando la posición inicial de la figura 2 el algoritmo la resolvió en 22 movimientos, manualmente puede hacerse en 20 movimientos, quizás en menos. El programa genera tableros aleatorios, como se mencionó, en todos los casos el algoritmo llegó a una solución. El número de movimientos necesarios para encontrar la solución varía de 20 a 70. Una optimización que probé después de ordenar el primer renglón fue mover el 4 y el 7 a su lugar de destino y hacer giros para acomodar el 5, 6 y 8. Con esta optimización el algoritmo toma 28 movimientos en llegar a la solución.

4.2 Solución Greedy al juego de "Mente Maestra"

El juego de mente maestra (Master Mind) se juega entre dos oponentes. Un jugador selecciona cuatro canicas de hasta seis colores diferentes y las mantiene ocultas del oponente. El segundo jugador debe adivinar el color y la posición en el tablero de las cuatro canicas en el menor número de intentos. Para esto hace su primera jugada colocando su primer intento de cuatro canicas. El primer jugador informa cuantas canicas están en la posición correcta y cuantas tienen el color correcto pero no la posición correcta. Para esto hace uso de pijas de color negro en el primer caso y de color blanco en el segundo. El segundo jugador selecciona otras canicas y hace su segundo intento y esto se repite hasta adivinar la combinación.

Debido a que el problema descansa en información secreta, la solución puede obtenerse de la teoría de juegos. Si embargo, el tamaño de las matrices involucradas lo convierten en un problema PN-completo. La complejidad de determinar si la solución es única o jugar de cualquier lado del tablero óptimamente esta abierta [8].

En [2] se da una solución basada en lo siguiente. Usaremos la siguiente notación para representar las canicas: B-Black, C-Cyan, , G=Green, R=Red, Y=Yellow, W=White. Los posibles resultados de un intento se resumen en la tabla 1:

Tabla 1. Posibles combinaciones de resultados.

caso	Pijas
0	Sin pijas
1	1 blanca
2	1 negra
3	2 blancas
4	1 blanca y 1 negra
5	2 negras
6	3 blancas
7	1 negra y 2 blancas
8	2 negras y 1 blanca
9	3 negras
10	4 blancas
11	1 negra y 3 blancas
12	2 negras 2 blancas
13	4 negras [solución]

El número de combinaciones posibles de las cuatro canicas es 6 x 6 x 6 x 6 = 1296. Un tablero con n colores y m canicas tendría m^n combinaciones diferentes. La primera jugada se puede ejecutar de cinco maneras diferentes, en términos del número de colores diferentes, esto es: BBBB, BBBC, BBCC, BBCG y BCGR. Después de la primera jugada el número de soluciones posibles se reduce, la tabla 2 tomada de [9] muestra todas las posibilidades.

Tabla 2. Posibles movimientos después del primer intento.

	Primer intento				
Primer resultado	BBBB	BBBC	BBCC	BBCG	BCGR
0	625	256	256	81	16
1	0	308	256	276	152
2	500	317	256	182	108
3	0	61	96	222	312
4	0	156	208	230	252
5	15	123	114	105	96
6	0	0	16	44	136
7	0	27	36	84	132
8	0	24	32	40	48
9	20	20	20	20	20
10	0	0	1	2	9
11	0	0	0	4	8
12	0	3	4	5	6
13	1	1	1	1	1

Por ejemplo, si el primer movimiento es BBBC y el resultado es el número 12, lo cual significa que dos están en su lugar y dos fuera de su lugar, solo hay 3 posibilidades, BBCB, BCBB y CBBB.

Note que el mejor movimiento es BBCC, ya que es el que tiene el valor pico más pequeño, 256. Todos los demás tienen valores pico más grandes, BBBB 625, BBBC 317, BBCG 1276 y BBCGR 312.

En 1993, Kenji Koyama y Tony W. Lai calcularon que la mejor estrategia usa un promedio de 5625/1296 = 4.340 movimientos (con un caso implicando 6 movimientos). En [2] se da una tabla con los 1275 casos que requiere cuando mucho 5 movimientos. Como ejemplo, del uso de la tabla esta el siguiente:

BBCC - 1 blanca, caso 1
CGRR - 1 blanca, caso 1
CYBY - 1 blanca, caso 1
BRWW - 2 blancas, caso 3
GWGB solución

El renglón de la tabla del que se obtiene l solución es:

```
BBCC[1].CGRR[1].CYBY[1].BRWW[3].GWGB[soln]
```

Solución propuesta

El algoritmo propuesto es extremadamente simple y consta de los siguiente pasos:

1. Determinar cuantas canicas de cada color existen en la secuencia objetivo.
2. Generar todas las combinaciones de los cuatro colores obtenidos hasta obtener la solución.

Por ejemplo, suponga que la cadena a adivinar es GRCG, los siguientes pasos se llevarían a cabo:

1. Determinar cuantas canicas de cada color existen en la secuencia objetivo

cadena de entrada	salida
BBBB	0 blancas 0 negras
CCCC	0 blancas 1 negras
GGGG	0 blancas 2 negras
RRRR	0 blancas 1 negras
YYYY	0 blancas 0 negras
WWWW	0 blancas 0 negras

Note que este paso puede terminar en el cuarto renglón. La siguiente entrada será CGGR.

2. Generar todas las combinaciones de los cuatro colores obtenidos hasta obtener la solución

CGGR	4 blancas 0 negras
CGRG	4 blancas 0 negras
CRGG	2 blancas 2 negras
GCGR	3 blancas 1 negras
GCRG	2 blancas 2 negras
GGCR	2 blancas 2 negras
GGRC	1 blancas 3 negras
GRCG	0 blancas 4 negras

Se han omitido los casos repetidos. En este caso la solución se obtiene en 12 pasos. Note que una vez obtenidos los colores que componen la solución solo es necesario señalar en que momento se presenta la solución, es decir, el número de pijas blancas y negras es irrelevante.

El algoritmo determina la solución a lo más en 30 pasos. 6 pasos para determinar los colores que intervienen en la solución y 4! = 24 pasos en generar todas las combinaciones con 4 colores diferentes.

Descripción del programa

El programa para implantar la solución fue escrito en Turbo Pascal. Define un objeto TmasterMind como sigue:

```
type
    TMasterMind = object
    private
      paso,numCorrectas,numBien:integer;
      Intento:array[1..30]of string[4];
    public
      procedure Init;
      procedure LeerObjetivo;
      procedure muestraJugada(n:integer);
      procedure leerMarcador;
      procedure juega;
      procedure MuestraResultado;
    end;
```

El método principal es juega. Como se indicó anteriormente, el primer paso que determina el número de canicas de cada color se interrumpe al obtener 4 negras. La segunda parte del método genera las combinaciones de los colores obtenidos. Para no repetir las combinaciones en caso de tener colores repetidos, cada combinación generada es almacenada en un arreglo y a cada paso se busca en el arreglo. El listado es el siguiente:

```
procedure TMasterMind.Juega;
var
  i,j,k,l,m,p,total:integer;
  ficha:string[4];
  probada:array[1..24]of string[4];
  YaProbada:boolean;
begin
  total:=0;
  ficha:='    ';
  writeln('--- Primera etapa---');
  repeat
    inc(paso);
    Intento[paso]:=col[paso]+col[paso]+col[paso]+col[paso];
    muestraJugada(paso);
    leerMarcador;
    if numCorrectas>0 then
      for j:=1 to numCorrectas do begin
        inc(total);
        ficha[total]:=Intento[paso][j];
        if total = 4 then break;
      end;
  until (paso=MaxColores)or(total=4);
  writeln('--- Segunda etapa---');
  p:=0;
  for i:=1 to 24 do
    Probada[i]:='';
  for i:=1 to 4 do
    for j:=1 to 4 do
      if i<>j then
        for k:=1 to 4 do
          if (k<>i)and(k<>j) then begin
            l:=10-(i+j+k);
            {busca la combinación para no repetirla}
            yaProbada:=false;
            for m:=1 to 24 do
              if (ficha[i]+ficha[j]+ficha[k]+
                  ficha[l])=probada[m] then
                yaProbada:=true;
            if not YaProbada then begin
              inc(paso);
              Intento[paso,1]:=ficha[i];
              Intento[paso,2]:=ficha[j];
              Intento[paso,3]:=ficha[k];
              Intento[paso,4]:=ficha[l];
              inc(p);
              probada[p]:=intento[paso];
              muestraJugada(paso);
```

```
            leerMarcador;
            if numCorrectas = 4 then exit;
          end;
        end;
end;
```

Los demás métodos y el programa principal se explican por sí mismos. Se anexa un listado del programa en el apéndice A.

4.3 Algoritmo de compresión de Huffman

La comunicación a través de Internet ha significado el flujo de grandísimas cantidades de información. Conforme la tecnología avanza las líneas de comunicación se hacen más rápidas y eficientes. Desgraciadamente la cantidad de información que se transmite crece a un ritmo similar o mayor. Esto supone que el uso de algoritmos de compresión de información eficientes, que generen un ahorro en la cantidad de datos a transmitir, siempre será necesario.

En 1950 David Huffman siendo aun estudiante del MIT desarrollo un algoritmo de compresión de datos que ha tenido un gran impacto en nuestros días.

Descripción del algoritmo

El algoritmo de Hoffman es un algoritmo Greedy (voraz). El algoritmo se basa en los siguientes puntos:

- Asigna códigos en bits más pequeños a los caracteres más frecuentes.
- Se debe conocer la frecuencia (al menos aproximada) de la aparición de cada carácter.
- Se debe enviar, junto con los datos, el código asignado y su longitud a cada carácter, para poder recuperar la información.

De acuerdo con estos puntos, los códigos de hoffman para cada lenguaje (humano o de programación) serán diferentes. Por ejemplo, en C los caracteres "{" o "}" aparecen con mucha frecuencia. Estos deberán tener códigos más breves.

Consideremos el siguiente ejemplo tomado de [1]. Sean los símbolos A, B, C y D y que se asignan códigos a estos símbolos como sigue:

Símbolo	Código
A	010
B	100
C	000
D	111

El mensaje ABACCDA se codificaría entonces como 01010001000000011101, esto es 16 bits. Si asignamos un código de dos bits, como el siguiente:

Símbolo	Código
A	00
B	01
C	10
D	11

El mensaje anterior se codifica como: 00010010101100, 14 bits. Si asignamos el siguiente código.

Símbolo	Código
A	0
B	110
C	10
D	111

El mensaje anterior se codifica como: 01100101011110, 13 bits. Si el mensaje fuera más largo el ahorro sería significativo. Note que en la última asignación cada carácter se reconoce de la siguiente manera, si el primer bit es 0, se tiene una A, si el primer bit es 1, se tiene B, C o D. Para distinguir entre estos tres caracteres debemos verificar el segundo bit, si es 0, se tiene C y si es 1, se tiene B o D. Por último si en el tercer bit se tiene un 0, se tiene una B sino una D.

Para encontrar la codificación óptima, empezamos a analizar los símbolos con menor frecuencia. En el ejemplo B y D tienen una frecuencia de 1. El código para B es 0 y para D es 1. Si juntamos estos símbolos la frecuencia de ocurrencia de B o D será de 2. El siguiente con menor frecuencia es C (2). Si lo combinamos con el símbolo BD nos dará una frecuencia de 4 para el símbolo CBD y sus códigos serán 0 para C y 1 para BD. Por último lo combinamos con A para obtener ACBD con códigos 0 para A y 1 para CBD.

Se puede utilizar un árbol binario para almacenar esta información. Cada nodo del árbol representa un símbolo, y cada hoja, un símbolo del alfabeto original. La figura 1 muestra el árbol binario construido por medio del ejemplo previo.

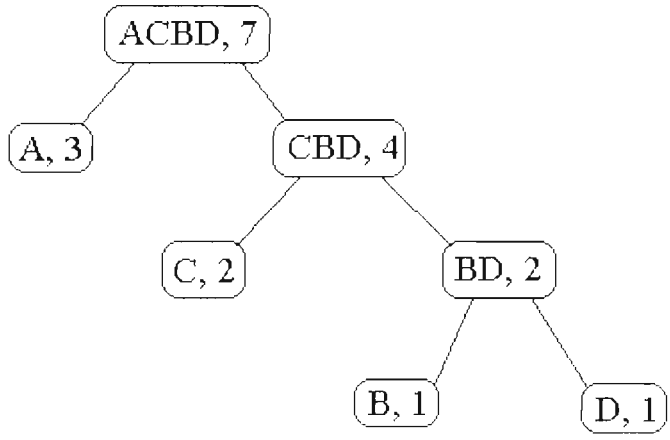


Figura 1. Árbol de Huffman.

Construcción del árbol

La creación de un árbol binario de búsqueda óptimo requiere de un tiempo exponencial. El algoritmo de Hoffman utiliza la siguiente estrategia.

1. Poner los n caracteres en n subárboles en una lista ordenada
2. Combinar los dos nodos con menor frecuencia en un árbol cuyo nodo raíz tenga la suma de las frecuencias de los dos hijos.
3. Insertar el árbol en la posición adecuada de la lista de árboles.
4. Repetir desde el paso 2 hasta que solo quede un nodo raíz.

El proceso necesariamente termina ya que a cada paso se baja en uno el número de nodos de la lista. El proceso de inserción requiere $O(\ln n)$ pasos. Como se van a ejecutar n iteraciones, el algoritmo tiene una complejidad de $O(n \ln n)$.

Codificación y Decodificación

Para codificar el mensaje se construye una tabla con una entrada por cada carácter del alfabeto con su código Hoffman. La construcción de la tabla requiere el recorrido recursivo del árbol agregando en cada hijo izquierdo un 0 y en cada hijo derecho un 1 a la secuencia de codificación.

La decodificación es simple. Comenzando con el primer bit, se decide si tomar el subárbol izquierdo o derecho, se toma el siguiente bit para determinar el siguiente subárbol. El proceso se continúa hasta llegar a una hoja, la cual contiene el carácter decodificado. El siguiente bit se toma como el primer bit para la siguiente etapa. De esta manera se continúa hasta agotar todos los bits de entrada.

En [10] se menciona el código de Huffman canónico. Este se define con las siguientes características:

- Los códigos más cortos tienen un valor numérico (si se agregan ceros a la derecha) más alto que los códigos largos.
- En códigos de la misma longitud los valores numéricos se incrementan con el orden alfabético.

Los códigos de Huffman canónicos tienen la misma longitud que los códigos normales. Existen algunas ventajas en usar códigos de Huffman canónicos.

- La primera regla garantiza que no más de ceil($\log_2$(tamaño alfabeto)) de los bits de la derecha pueden diferir de cero.
- La primera regla permite una decodificación eficiente.
- Ambas reglas juntas permiten la reconstrucción del código sabiendo solo la longitud del código de cada símbolo.

Aplicaciones

Explorando la red pueden encontrarse gran variedad de aplicaciones del algoritmo de Hoffman. Estas aplicaciones se dan desde la codificación de texto, audio (MP3), gráficos (JPEG), etc.

CAPITULO 5

CONCLUSIONES

Conclusiones

En el primer capitulo se vio que la operación de inserción es considerablemente más eficiente en listas que en arreglos debido a que solo se mueven los apuntadores y no los datos.
Los árboles-B son estructuras de búsqueda eficientes que pueden almacenar gran cantidad de información. La inserción y el borrado de nodos son procesos complejos. En el capitulo 1 se mostró gráficamente como se llevan a cabo dichos procesos. Solo se describió brevemente el proceso de inserción, pero puede intuirse que el proceso de borrado es aún más complejo.

La búsqueda mediante una tabla de dispersión es en extremo eficiente, es por eso que se utiliza mucho en compiladores y ensambladores donde se requiere una búsqueda rápida de identificadores, etiquetas y palabras reservadas. En el capitulo 2 he mostrado algunas técnicas de manejo de tablas de dispersión de tamaño fijo. Incluí algunos algoritmos en C y Pascal. El simulador de [4] ilustra claramente el comportamiento de las tablas de dispersión de tamaño fijo.

Los grafos tienen muchas aplicaciones en la resolución de problemas en cómputo. Los algoritmos para manejo de grafos son en general complicados y costosos. En el capitulo 3 se han definido algunos conceptos sobre grafos dirigido y no dirigidos.

El algoritmo de Dijkstra revisado también en el capitulo 3 es fácil de entender e implantar. La implementación en Delphi fue simple. La ventaja de haber utilizado un hilo adicional es que puede detenerse su ejecución sin perder los valores de las variables locales y continuarlo después desde donde se quedo.

La solución presentada al problema del rompecabezas en el capitulo 4 es general. Puede resolver el problema desde cualquier posición inicial. El programa utiliza una heurística que hace uso de la forma de resolver el problema manualmente. Puede aplicarse a tableros más grandes, pero implicaría escribir las cadenas de movimientos para ese caso además de agregar sentencias `if` al método `resolver`.

La solución presentada al problema de Mente Maestra es simple, aunque no es óptima. Fácilmente se puede generalizar a un número más grande de colores o de canicas. La solución de un tablero con n colores y m canicas requiere de n pasos para determinar los colores de las canicas y $m!$ pasos para determinar la combinación ganadora, en el peor de los casos. Si m es grande (>10) el tiempo puede ser prohibitivo, pero una tabla como la mostrada en [9] puede requerir de un esfuerzo sobrehumano.

Finalmente En [11] se encuentra un simulador muy didáctico que muestra la construcción de un código de Huffman para varios conjuntos de datos de entrada. El simulador muestra el proceso de creación de árbol, y su uso para construir la tabla de códigos y la decodificación de secuencias de bits.
Considero que estos temas son de gran ayuda para la comprensión y solución de algoritmos de programación avanzada.

Referencias

[1] Aarón M Tenenbaum, Yedidyah Langsam, Moshe A. Augenstein, *Estructuras de datos en C*, Prentice may, 1993.

[2] http://cis.stvincent.edu/carlsond/swdesign/btree/btree.html

[3] Niklaus Wirth, *Algoritmos + Estructuras de Datos = Programas*. Ediciones Castillo. 1976.

[4] http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/hash_tables.html

[5] http://burtleburtle.net/bob/hash/perfect.html

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, 2a. edición.

[7] http://mathworld.wolfram.com/15Puzzle.html, 3de junio de 2003.

[8] http://www.ics.uci.edu/~eppstein/cgt/hard.html#mastermind

[9] http://www.tnelson.demon.co.uk/mastermind/

[10] http://www.compressconsult.com/huffman/

[11] http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/huffman.html

APENDICE A

Árboles-B

```
program BTrees;

uses crt;

const ORDEN = 5;

type
  TLlave= string[8];
  PNodo = ^TNodo;
  TNodo = record
    numHijos : 0..ORDEN;
    padre : PNodo;
    indice : integer;
    llave : array [1..ORDEN-1] of TLlave;
    p : array [1..ORDEN] of PNodo;
  end;
  TArbolB = object
  private
    raiz:PNodo;
  public
    function buscar(arbol:PNodo;Llave:TLlave;
                    var posicion:integer;var
Encontrado:boolean):PNodo;
    function BuscaNodo(p:PNodo;Llave:TLlave):integer;
    function crearArbol(Llave:TLlave):PNodo;
    procedure Insertar(llave1: TLlave; s : PNodo; posicion :
integer);
    procedure split(nd : PNodo; pos : integer;
                    nuevaLlave:TLlave;var
nuevoNodo,nd2:PNodo;
                    var mediaLlave:TLlave);
    procedure insertaNodo(nd : PNodo;pos:integer;
                    nuevaLlave:TLlave;nuevoNodo:PNodo);
    procedure
copy(nd1:PNOdo;primera,ultima:integer;nd2:PNOdo);
    procedure InsertaLlave(Llave:TLlave);
    procedure recorre(arbol:PNodo;n:integer);
    destructor destruir;
  end;

var b:TArbolB;
    TamanyoNodo,NumNodos,NumLlaves,Nivel:integer;
```

```
procedure TArbolB.InsertaLlave(Llave:TLlave);
var pos:integer;
    Encontrado:boolean;
    p:PNodo;
begin
  p:=buscar(raiz,Llave,pos,Encontrado);
  if not Encontrado then
    Insertar(Llave,p,pos);
end;

procedure TArbolB.recorre(arbol:PNodo;n:integer);
var i,nh:integer;
begin
  if (arbol<>NIL) then begin
    nh:=arbol^.numHijos;
    for i:=1 to nh-1 do begin
      recorre(arbol^.p[i],n+1);
      textcolor(15-n);
      write(i:2,arbol^.Llave[i]:8);
{      readkey;}
    end;
    recorre(arbol^.p[nh],n+1);
  end;
end;

destructor TArbolB.destruir;
  procedure borraNodo(arbol:PNodo);
  var i,nh:integer;
  begin
    if (arbol<>NIL) then begin
      nh:=arbol^.numHijos;
      for i:=1 to nh do begin
        borraNodo(arbol^.p[i]);
        if arbol^.p[i]<>nil then
          dispose(arbol^.p[i]);
      end;
    end;
  end;
begin
  borraNodo(raiz);
end;

function TArbolB.BuscaNodo(p:PNodo;Llave:TLlave):integer;
{esta función localiza la llave más pequeña en un nodo mayor
o igual
 que el argumeto de la búsqueda}
var i:integer;
```

```
begin
  BuscaNodo:=p^.numHijos;
  for i:=1 to p^.numHijos-1 do
    if Llave<=p^.Llave[i] then begin
      BuscaNodo:=i;
      exit;
    end;
end;

function TArbolB.buscar(arbol:PNodo;Llave:TLlave;
                        var posicion:integer;var
Encontrado:boolean):PNodo;
{Esta función es la encargada de buscar una llave en el
árbol.
 Regresa el apuntador al nodo donde está la llave y la
posición dentro
 del arreglo de llaves. Si la llave no se encuentra, regresa
la última
 hoja visitada.}
var p,q:PNodo;i:integer;
begin
  q:=NIL;
  p:=arbol;
  while p<>NIL do begin
    i:=BuscaNodo(p,Llave);
    q:=p;
    if (i<p^.numHijos)and(Llave=p^.Llave[i])then begin
      encontrado:=true;
      posicion:=i;
      buscar:=p;
      exit;
    end;
    p:=p^.p[i];
  end;
  Encontrado:=false;
  posicion:=i;
  buscar:=q;
end;

function TArbolB.crearArbol(Llave:TLlave):PNodo;
{Crea un árbol con un solo nodo y regresa un apuntador al
mismo. }
var p:PNodo;
    i:integer;
begin
  inc(Nivel);
  inc(numNodos);
```

```
  new(p);
  with p^ do begin
    padre:=NIL;
    numHijos:=2;
    indice:=0;
  end;
  p^.llave[1]:=Llave;
  for i:=2 to ORDEN-1 do
    p^.llave[i]:='';
  for i:=1 to ORDEN do
    p^.p[i]:=NIL;
  crearArbol:=p;
end;

procedure TArbolB.Insertar(llave1: TLlave; s : PNodo;
posicion : integer);
{Inserta una llave en un árbol-B.}
var nd, nd2, nuevoNodo, f : PNodo;
    pos : 0..ORDEN;
    nuevaLlave,mediaLlave:TLlave;
begin
  inc(NumLlaves);
  nd:=s;
  pos:=posicion;
  nuevoNodo:=NIL;
  nuevaLlave:=Llave1;
  f:=nd^.padre;
  while(f <> NIL)and(nd^.numHijos=ORDEN)do begin
    split(nd,pos,nuevaLlave,nuevoNodo,nd2,mediaLlave);
    nuevoNodo:=nd2;
    pos:=nd^.indice;
    nd:=f;
    f:=nd^.padre;
    nuevaLlave:=mediaLlave;
  end;
  if nd^.numHijos<ORDEN then
    insertaNodo(nd,pos,nuevaLlave,nuevoNodo)
  else begin
    split(nd,pos,nuevaLlave,nuevoNodo,nd2,mediaLlave);
    raiz:=crearArbol(mediaLlave);
    raiz^.p[1]:=nd;
    raiz^.p[2]:=nd2;
    nd^.indice:=1;      {//actualiza el indice}
    nd^.padre:=raiz;    {//actualiza el padre del hijo}
    nd2^.indice:=2;      {//actualiza el indice        }
    nd2^.padre:=raiz;    {//actualiza el padre del hijo }
  end;
```

```
end;

procedure TArbolB.insertaNodo(nd : PNodo;pos:integer;
                    nuevaLlave:TLlave;nuevoNodo:PNodo);
{Inserta una nueva llave dentro de la posición pos de un nodo
no lleno.
 nuevoNodo apunta a un subárbol que debe insertarse a la
derecha de la
 nueva llave. Las llaves restantes y los subárboles en las
posiciones
 pos o mayores se recorren una posición.}
var i:integer;
begin
  for i:=nd^.numHijos downto pos+1 do begin
    nd^.p[i+1]:=nd^.p[i];
    nd^.llave[i]:=nd^.llave[i-1];
    if nd^.p[i+1]<>NIL then
      nd^.p[i+1]^.indice:=i+1; {//actualiza el indice}
  end;
  nd^.p[pos+1]:=nuevoNodo;
  if nd^.p[pos+1]<>NIL then begin
    nd^.p[pos+1]^.indice:=pos+1; {//actualiza el indice  }
    nd^.p[pos+1]^.padre:=nd;   {//actualiza el padre del
hijo}
  end;
  nd^.llave[pos]:=nuevaLlave;
  inc(nd^.numHijos);
end;

procedure TArbolB.split(nd : PNodo; pos : integer;
                    nuevaLlave:TLlave;var
nuevoNodo,nd2:PNodo;
                    var mediaLlave:TLlave);
{divide en dos a un nodo. Las n/2 llaves menores quedan en el
nodo nd,
 y las demás son colocadas en un nuevo nodo nd2.}
begin
{// crea un nuevo nodo para la mitad derecha}
  inc(numNodos);
  new(nd2);
  if pos>ORDEN div 2+1 then begin
{// nuevaLlave pertenece a nd2}
    copy(nd,ORDEN div 2+2,ORDEN-1,nd2);
    insertaNodo(nd2,pos-ORDEN div 2-1,nuevaLlave,nuevoNodo);
    nd^.numHijos:=ORDEN div 2+1;
    mediaLlave:=nd^.Llave[ORDEN div 2+1];
  end;
```

```
  if pos=ORDEN div 2+1 then begin
{    // nuevaLlave es la de en medio}
    copy(nd,ORDEN div 2+1,ORDEN-1,nd2);
    nd^.numHijos:=ORDEN div 2+1;
    nd2^.p[1]:=NuevoNodo;
    mediaLlave:=nuevaLlave;
  end;
  if pos<ORDEN div 2+1 then begin
{    // nuevaLlave pertenece a nd}
    copy(nd,ORDEN div 2+1,ORDEN-1,nd2);
    nd^.numHijos:=ORDEN div 2;
    mediaLlave:=nd^.Llave[ORDEN div 2];
    insertaNodo(nd,pos,nuevaLlave,nuevoNodo);
  end;
end;

procedure
TArbolB.copy(nd1:PNOdo;primera,ultima:integer;nd2:PNOdo);
{Copia las llaves del nodo nd1 desde primera hasta última en
las llaves
 de nd2 de la 1 hasta ultima-primera+1. También debe
actualizar el
 apuntador al padre de cada hijo que se copie y el indice.}
var i,numLlaves:integer;
begin
  if ultima<primera then
    nd2^.numHijos:=1
  else begin
    numLlaves:=ultima-primera+1;
    for i:=primera to ultima do
      nd2^.Llave[i-primera+1]:=nd1^.Llave[i];
    for i:=primera to ultima+1 do begin
      nd2^.p[i-primera+1]:=nd1^.p[i];
      if nd2^.p[i-primera+1]<>nil then begin
        nd2^.p[i-primera+1]^.padre:=nd2; {
//actualiza el padre del hijo}
        nd2^.p[i-primera+1]^.indice:=i-primera+1;
{//actualiza el indice          }
      end;
    end;
    nd2^.numHijos:=numLlaves+1;
  end;
end;

procedure Leer;
var cad:string[10];
    f:text;
```

```
    pos,i:integer;
    Encontrado:boolean;
    p:PNodo;
begin
  textcolor(white);
  assign(f,'test.txt');
  reset(f);
  while not eof(f) do begin
    readln(f,cad);
    if b.raiz=nil then begin
      inc(NumLlaves);
      b.raiz:=b.CrearArbol(cad);
    end
    else
      b.InsertaLlave(cad);
  end;
  close(f);
  repeat
    clrscr;
    b.recorre(b.raiz,0);
    writeln;
    for i:=0 to 3 do begin
      textcolor(15-i);
      write('nivel ',i,' ');
    end;
    writeln;
    textcolor(15);
    write('teclee clave a buscar:');
    readln(cad);
    p:=b.buscar(b.raiz,cad,pos,Encontrado);
    if Encontrado then
      writeln(cad,' en posici$n: ',pos)
    else
      writeln('No encontre a ',cad,' en el  rbol.');
    readkey;
  until cad='fin';
  writeln('N£mero de llaves: ',numLlaves);
  writeln('N£mero de nodos: ',numNodos);
  writeln('N£mero de niveles: ',nivel);
  writeln('Tamaño del nodo: ',TamanyoNodo,' bytes');
  writeln('Memoria utilizada total: ',TamanyoNodo*numNodos,'
bytes');
  writeln('Memoria utilizada en datos:
',(4*sizeof(Tllave))*numNodos,' bytes');
  writeln('Llaves por nodo: ',numLlaves/numNodos:8:4);
  readkey;
end;
```

```
begin
  clrscr;
  numNodos:=0;
  tamanyoNodo:=sizeof(TNodo);
  NumLlaves:=0;
  Nivel:=0;
  b.raiz:=nil;
  Leer;
  b.destruir;
end.
```

Programa Dijkstra

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs,
  ExtCtrls, Buttons, StdCtrls, Menus, Spin;

const MaxNodos = 16;
      INF = MaxInt DIV 2;

type
  Nodo = 0..MaxNodos;
  TipoDist = array[1..MaxNodos]of integer;
  conjuntoNodos = set of Nodo;
  TGrafo = object
  private
    Coste : array[1..MaxNodos,1..MaxNodos]of integer;
    NumNodos : integer;
  public
    procedure PonCoste(i,j:Nodo;c:integer);
    function DameCoste(i,j:Nodo):integer;
    procedure PonNumNodos(n:Nodo);
    function DameNumNodos:Nodo;
    constructor init;
  end;
  TForm1 = class(TForm)
    Panel1: TPanel;
    Image1: TImage;
    Memo1: TMemo;
    Button1: TButton;
    MainMenu1: TMainMenu;
    Archivo1: TMenuItem;
    Cargargrafo1: TMenuItem;
    N1: TMenuItem;
    Salir1: TMenuItem;
    OpenDialog1: TOpenDialog;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Memo2: TMemo;
    Button2: TButton;
    Label4: TLabel;
    SpinEdit1: TSpinEdit;
```

```
    SpinEdit2: TSpinEdit;
    Acercade1: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure Cargargrafo1Click(Sender: TObject);
    procedure Salir1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Acercade1Click(Sender: TObject);
  private
    { Private declarations }
    G:TGrafo;
  public
    { Public declarations }
    function Min(a,b:integer):integer;
    procedure Dijkstra(Comienzo,Final:Nodo;n:integer;var
Dist:TipoDist);
    procedure dibuja;
    procedure hecho(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

uses math, Unit2, Unit3;

constructor TGrafo.init;
begin
  PonNumNodos(0);
end;

procedure TGrafo.PonCoste;
begin
  Coste[i,j]:=c;
end;

function TGrafo.DameCoste(i,j:Nodo):integer;
begin
  DameCoste:=Coste[i,j];
end;

procedure TGrafo.PonNumNodos;
begin
  NumNodos:=n;
end;
```

```
function TGrafo.DameNumNodos;
begin
  DameNumNodos:=NumNodos;
end;

procedure
Arrow(x1,y1,x2,y2:double;color:TColor;canvas:TCanvas);
var a,l,d,dx,dy,x3,y3,x4,y4,t:double;
    points:array[1..3]of TPoint;
begin
  with Canvas do begin
    Points[1].x:=round(x2);
    Points[1].y:=round(y2);
    pen.Color:=color;
    MoveTo(round(x1),round(y1));
    Moveto(round(x2),round(y2));
    dx:=x2-x1;
    dy:=y2-y1;
    l:=sqrt(dx*dx+dy*dy);
    if l > 0 then begin
      if l<10 then d:=l else d:=10;
      x3:=-d;y3:=d/3;
      x4:=-d;y4:=-d/3;
      a:=arctan2(dy,dx);
      t:=x3*cos(a)-y3*sin(a);
      y3:=x3*sin(a)+y3*cos(a);
      x3:=t;
      t:=x4*cos(a)-y4*sin(a);
      y4:=x4*sin(a)+y4*cos(a);
      x4:=t;
      x3:=x3+x2;
      y3:=y3+y2;
      x4:=x4+x2;
      y4:=y4+y2;
      MoveTo(round(x3),round(y3));
      LineTo(round(x2),round(y2));
      LineTo(round(x4),round(y4));
    end;
  end;
end;

procedure TForm1.dibuja;
{Dibuja el grafo}
var i,j,x1,y1,x2,y2,NumNodos:integer;
begin
  with Image1.Canvas do begin
```

```
    brush.Style:=bsSolid;
    brush.color:=clWhite;
    Pen.Color:=clWhite;
    rectangle(0,0,image1.width,image1.height);
    Pen.Color:=clBlack;
    brush.Style:=bsClear;
    NumNodos:=G.DameNumNodos;
    for i:=1 to MaxNodos do
      for j:=1 to MaxNodos do
        if G.DameCoste(i,j)<>INF then begin
          x1:=250+round(200*sin(2*i*pi/NumNodos));
          y1:=250+round(200*cos(2*i*pi/NumNodos));
          x2:=250+round(200*sin(2*j*pi/NumNodos));
          y2:=250+round(200*cos(2*j*pi/NumNodos));
          moveto(x1,y1);
          Lineto(x2,y2);
          Arrow(x1,y1,x1+(x2-x1)*9/10,y1+(y2-y1)*9/10,
               clBlack,Image1.canvas);
          Font.Color:=clRed;
          TextOut(x1+(x2-x1) div 4,y1+(y2-y1) div 4,
             IntToStr(G.DameCoste(i,j)));
        end;
    Font.Color:=clBlue;
    for i:=1 to NumNodos do begin
        x1:=250+round(200*sin(2*i*pi/NumNodos));
        y1:=250+round(200*cos(2*i*pi/NumNodos));
        brush.Style:=bsSolid;
        Ellipse(x1-12,y1-12,x1+12,y1+12);
        brush.Style:=bsClear;
        TextOut(x1-6,y1-6,IntToStr(i));
      end;
  end;
end;

function TForm1.Min(a,b:integer):integer;
begin
  if a<b then
    Min:=a
  else
    Min:=b;
end;

procedure TForm1.Dijkstra (Comienzo, Final:Nodo;
     n:integer; var Dist:TipoDist);
var i, u : Nodo;
    DistMin:integer;
    S,OldS:conjuntoNodos;
```

```
begin
  for i:=1 to n do
    Dist[i]:=INF;
  Dist[Comienzo]:=0;
  S:=[Comienzo];
  u:=Comienzo;
  for i:=1 to n do
    if not(i in S) then
      Dist[i]:=min(Dist[i],Dist[u]+G.DameCoste(u,i));
  OldS:=S;
  repeat
    DistMin:=INF;
    for i:=1 to n do
      if not (i in S) and (Dist[i]<DistMin) then begin
        DistMin:=Dist [i];
        u:=i;
      end;
    S:=S+[u];
    if S=OldS then begin
      ShowMessage('No se puede ir del nodo '
                  +intTOStr(comienzo)+' al nodo
'+intTOStr(final));
      exit;
    end
    else
      OldS:=S;
    for i:=1 to n do
      if not(i in S) then
        Dist[i]:=min(Dist[i],Dist[u]+G.DameCoste(u,i));
  until u = Final;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  Bitmap.Width := Form1.ClientWidth-Panel1.ClientWidth;
  Bitmap.Height := Form1.ClientWidth;
  Image1.Picture.Graphic := Bitmap;
  g.init;
end;

procedure TForm1.Cargargrafo1Click(Sender: TObject);
var f:textfile; i,j,m:integer;
begin
  if OpenDialog1.Execute then begin
```

```
      assignfile(f,OpenDialog1.FileName);
      reset(f);
      Memo1.Clear;
      Memo1.Lines.Add('Nodo1 Nodo2 Coste');
      for i:=1 to MaxNodos do
        for j:=1 to MaxNodos do
          G.PonCoste(i,j,INF);
      while not eof(f) do begin
        readln(f,i,j,m);
        Memo1.Lines.Add(Format('%4d %4d %4d',[i,j,m]));
        G.PonCoste(i,j,m);
        if G.DameNumNodos<i then
          G.PonNumNodos(i);
        if G.DameNumNodos<j then
          G.PonNumNodos(j);
      end;
      closefile(f);
      if Memo1.Lines.Count>=15 then
        Memo1.ScrollBars:=ssVertical
      else
        Memo1.ScrollBars:=ssNone;
      if Memo2.Lines.Count>=10 then
        Memo2.ScrollBars:=ssVertical
      else
        Memo2.ScrollBars:=ssNone;
      with SpinEdit1 do begin
        MinValue:=1;
        MaxValue:=g.DameNumNodos;
        Value:=1;
        Enabled:=true;
      end;
      with SpinEdit2 do begin
        MinValue:=1;
        MaxValue:=g.DameNumNodos;
        Value:=g.DameNumNodos;
        Enabled:=true;
      end;
      dibuja;
    end;
end;

procedure TForm1.Salir1Click(Sender: TObject);
begin
  application.Terminate;
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```
{este evento de botón construye el hilo de simulación.}
var i,j:integer;
begin
  i:=SpinEdit1.Value;
  j:=SpinEdit2.Value;
  if i=j then
    ShowMessage('Los nodos deben ser diferentes')
  else begin
    Button1.Enabled:=false;
    Button2.Enabled:=false;
    // crea hilo
    with DijstraThread.Create(Form2.Button1,Form2.Button2,
      Form2.memo1,g,i,j) do
      OnTerminate:=hecho;
    Form2.Show;
  end;
end;

procedure TForm1.hecho(Sender: TObject);
begin
//  showmessage('Hilo destruido');
end;

procedure TForm1.Button1Click(Sender: TObject);
{Este evento de botón despliega en el memo inferior
 el resultado de la ejecusión del algoritmo de Dijkstra.}
var i,j,k:integer; d:TipoDist;
begin
  i:=SpinEdit1.Value;
  j:=SpinEdit2.Value;
  if i=j then
    ShowMessage('Los nodos deben ser diferentes')
  else begin
    Dijkstra (i,j,g.DameNumNodos,d);
    memo2.Clear;
    for k:=1 to g.DameNumNodos do
      if d[k]<>INF then
        Memo2.Lines.Add('nodo: '+intToStr(k)+
          ' dist= '+intToStr(d[k]))
      else
        Memo2.Lines.Add('nodo: '+intToStr(k)+
          ' dist= INFINITO');
  end;
end;

procedure TForm1.Acercade1Click(Sender: TObject);
begin
```

```
  ShowMessage('Algoritmo de Dijkstra'+#13+#13+
              'Héctor E. Medellín Anaya'+#13+
              'Araceli Arevalo Ramírez'+#13+
              'Mayo de 2003');
end;

end.
```

Manejo de hilos

```
unit Unit3;

interface

uses
  Classes,stdctrls,Unit1,SysUtils,Unit2;

type
  DijstraThread = class(TThread)
  private
    { Private declarations }
    m:TMemo;
    B1:TButton;
    B2:TButton;
    g:TGrafo;
    Dist:TipoDist;
    u : Nodo;
    DistMin:integer;
    S:conjuntoNodos;
    Comienzo, Final:Nodo;
  protected
    procedure Execute; override;
  public
    procedure muestra;
    constructor
create(b,d:TButton;m1:TMemo;g1:TGrafo;C,F:Nodo);
    procedure ButClick(Sender: TObject);
    procedure FinalizarClick(Sender: TObject);
  end;

implementation


{ Important: Methods and properties of objects in VCL can
only be used in a
  method called using Synchronize, for example,
```

```
      Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure DijstraThread.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }

{ DijstraThread }

constructor
DijstraThread.Create(b,d:TButton;m1:TMemo;g1:TGrafo;C,F:Nodo)
;
{Crea el hilo de simulación. El hilo se crea dormido.}
begin
  b1:=b; //botón siguiente
  b2:=d; //botón terminar
  m:=m1; //memo para el despliegue
  FreeOnTerminate := True;
  b1.OnClick:=butClick;  //define los eventos de los botones
  b2.OnClick:=FinalizarClick;
  comienzo:=c;
  Final:=f;
  g:=g1;
  inherited Create(false);
end;

procedure DijstraThread.ButClick(Sender: TObject);
{despierta al hilo}
begin
  resume;
end;

procedure DijstraThread.FinalizarClick(Sender: TObject);
{solicita destruir al hilo}
begin
  terminate;
  resume;
end;

procedure DijstraThread.muestra;
{Muestra el progreso de la simulación}
var cad:string; j:integer;
begin
  m.Lines.Add('u = '+IntToStr(u));
  cad:='S = {';
```

```
  for j:=1 to G.DameNumNodos do
    if j in s then
      cad:=cad+IntToStr(j)+',';
  delete(cad,length(cad),1);
  m.Lines.Add(cad+'}');
  cad:='Dist = [';
  for j:=1 to G.DameNumNodos do
    if dist[j]=INF then
      cad:=cad+'INF,'
    else
      cad:=cad+IntToStr(dist[j])+',';
  delete(cad,length(cad),1);
  cad:=cad+']';
  m.Lines.Add(cad);
end;

procedure DijstraThread.Execute;
{ Procedimiento Dijkstra modificado para hacer pausas a cada
paso. }
var i:integer;
    hecho:boolean;
begin
  m.Clear;
  m.Lines.Add('Comienzo = '+IntToStr(comienzo)+
              ' Final = '+IntToStr(final));
  for i:=1 to g.DameNumNodos do
    Dist[i]:=INF;
  Dist[Comienzo]:=0;
  S:=[Comienzo];
  u:=Comienzo;
  m.Lines.Add('Inicio del algoritmo de Dijkstra');
  muestra;
  suspend;
  hecho:=terminated;
  if not hecho then begin
    m.Lines.Add('Distancias a los nodos adyacentes al nodo
'+IntToStr(u));
    for i:=1 to g.DameNumNodos do
      if not(i in S) then begin
        Dist[i]:=Form1.Min(Dist[i],Dist[u]+G.DameCoste(u,i));
      end;
    muestra;
    suspend;
    hecho:=terminated;
    m.Lines.Add('Lazo REPEAT...');
  end;
  if not hecho then
```

```
  repeat
    DistMin:=INF;
    for i:=1 to g.DameNumNodos do
      if not(i in S) and (Dist[i]<DistMin) then begin
        DistMin:=Dist[i];
        u:=i;
      end;
    S:=S+[u];
    for i:=1 to g.DameNumNodos do
      if not(i in S) then
        Dist[i]:=Form1.min(Dist[i],Dist[u]+G.DameCoste(u,i));
    muestra;
    suspend;
    hecho:=terminated;
  until (u = Final)or hecho;
  m.Lines.Add('TERMINADO');
  if not hecho then
    suspend;
  Form1.Button1.Enabled:=true;
  Form1.Button2.Enabled:=true;
  Form2.Hide;
end;

end.
```

Puzzle 9

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs,
  StdCtrls, Buttons, ExtCtrls;

type
  TSecuencia=string[20];
  TTablero=array[1..3,1..3]of integer;
  TPuzzle = object
  private
    solucion:string[200];
    movimientos:integer;
    tablero,tableroinicial:TTablero;
    hecho:boolean;
  public
    procedure init;
    procedure dibuja(canvas:TCanvas);
    procedure cambia(var a,b:integer);
    procedure arriba;
    procedure abajo;
    procedure derecha;
    procedure izquierda;
    procedure resolver;
    procedure desordenar;
    procedure secuencia(s:TSecuencia);
    function resuelto:Boolean;
  end;
  TForm1 = class(TForm)
    Label1: TLabel;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Timer1: TTimer;
    procedure FormPaint(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button:
TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
```

```
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
    Puzzle:TPuzzle;
    n:integer;
    t:string[200];
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function TPuzzle.resuelto:Boolean;
var i,j:integer;
begin
  resuelto:=true;
  for i:=1 to 3 do
   for j:=1 to 3 do
     if 3*(i-1)+j<>9 then
       if tablero[i,j]<>3*(i-1)+j then
         resuelto:=false
       else
     else
       if tablero[i,j]<>0 then
         resuelto:=false;
end;

procedure TPuzzle.secuencia(s:TSecuencia);
var i:integer;
begin
 hecho:=false;
 for i:=1 to length(s) do begin
   if resuelto then begin
     hecho:=true;
     exit;
   end;
   solucion:=solucion+s[i];
   case s[i] of
     'i':izquierda;
     'd':derecha;
     'a':arriba;
     'b':abajo;
```

```
    end;
  end;
end;

procedure TPuzzle.arriba;
var i,j:integer;
begin
   for i:=2 to 3 do
    for j:=1 to 3 do
      if tablero[i,j]=0 then begin
        cambia(tablero[i,j],tablero[i-1,j]);
        exit;
      end;
end;

procedure TPuzzle.abajo;
var i,j:integer;
begin
   for i:=1 to 2 do
    for j:=1 to 3 do
      if tablero[i,j]=0 then begin
        cambia(tablero[i,j],tablero[i+1,j]);
        exit;
      end;
end;

procedure TPuzzle.izquierda;
var i,j:integer;
begin
   for i:=1 to 3 do
    for j:=2 to 3 do
      if tablero[i,j]=0 then begin
        cambia(tablero[i,j],tablero[i,j-1]);
        exit;
      end;
end;

procedure TPuzzle.derecha;
var i,j:integer;
begin
   for i:=1 to 3 do
    for j:=1 to 2 do
      if tablero[i,j]=0 then begin
        cambia(tablero[i,j],tablero[i,j+1]);
        exit;
      end;
end;
```

```
procedure TPuzzle.init;
begin
  movimientos:=0;
  tablero[1,1]:=5;tablero[1,2]:=1;tablero[1,3]:=2;
  tablero[2,1]:=6;tablero[2,2]:=4;tablero[2,3]:=3;
  tablero[3,1]:=8;tablero[3,2]:=7;tablero[3,3]:=0;
  tableroinicial:=tablero;
end;

procedure TPuzzle.dibuja(canvas:TCanvas);
var i,j:integer;
begin
  with canvas do begin
    Font.Size:=20;
    Font.name:='Times New Roman';
    for i:=1 to 3 do
      for j:=1 to 3 do begin
        rectangle(100+50*(j-1)-15,100+50*(i-1)-5,
                  100+50*(j-1)+30,100+50*(i-1)+40);
        if tablero[i,j]<>0 then
          TextOut(100+50*(j-1),100+50*(i-
1),IntToStr(tablero[i,j]));
      end;
  end;
end;

procedure TPuzzle.cambia(var a,b:integer);
var c:integer;
begin
  c:=a; a:=b; b:=c;
end;

procedure TPuzzle.resolver;
var sec:TSecuencia;
begin
  {colocar vacío en el centro}
  sec:='';
  if tablero[2,2]<>0 then begin
    if tablero[1,1]=0 then sec:='db'
    else if tablero[1,3]=0 then sec:='ib'
    else if tablero[3,1]=0 then sec:='da'
    else if tablero[3,3]=0 then sec:='ia'
    else if tablero[1,2]=0 then sec:='b'
    else if tablero[2,1]=0 then sec:='d'
    else if tablero[2,3]=0 then sec:='i'
    else if tablero[3,2]=0 then sec:='a';
```

```
  secuencia(sec);
end;
{colocar 1 en [1,1]}
if tablero[1,1]<>1 then begin
  sec:='';
  if tablero[1,2]=1 then sec:='iadb'
  else if tablero[1,3]=1 then sec:='addbiiadb'
  else if tablero[2,1]=1 then sec:='aibd'
  else if tablero[2,3]=1 then sec:='daibiadb'
  else if tablero[3,1]=1 then sec:='ibdaaibd'
  else if tablero[3,2]=1 then sec:='biadaibd'
  else if tablero[3,3]=1 then sec:='bdaibiadaibd';
  secuencia(sec);
end;
{colocar 2 en [1,2]}
if tablero[1,2]<>2 then begin
  sec:='';
  if tablero[1,3]=2 then sec:='adbi'
  else if tablero[2,1]=2 then sec:='iadbdaiibdadbi'
  else if tablero[2,3]=2 then sec:='daib'
  else if tablero[3,1]=2 then sec:='biadbdaaib'
  else if tablero[3,2]=2 then sec:='bdaaib'
  else if tablero[3,3]=2 then sec:='dbiadaib';
  secuencia(sec);
end;
{colocar 3 en [1,3]}
if tablero[1,3]<>3 then begin
  sec:='';
  if tablero[2,1]=3 then sec:='iadbdaiibd'
  else if tablero[2,3]=3 then sec:='iaddbiaibd'
  else if tablero[3,1]=3 then sec:='biadbiaadbdaiibd'
  else if tablero[3,2]=3 then sec:='biaadbdaiibd'
  else if tablero[3,3]=3 then sec:='dbiaiaddbiaibd';
  secuencia(sec);
end;
{colocar 5 en [2,1]} //cambiar 5 por 4 para optimizar
if hecho then exit;
if tablero[2,1]<>5 then begin
  sec:='';
  if tablero[3,1]=5 then sec:='ibda'
  else if (tablero[3,2]=5)or (tablero[3,2]=5) then
    sec:='biad'
  else if (tablero[3,3]=5) then sec:='bdaibiad'
  else if (tablero[2,3]=5) then sec:='dbiiad';
  secuencia(sec);
end;
{colocar 4 en [3,1]} //cambiar 4 por 7 para optimizar
```

```
  if hecho then exit;
  if tablero[3,1]<>4 then begin
    sec:='';
    if tablero[3,2]=4 then sec:='ibddaiibdadbiiad'
    else if (tablero[3,3]=4) then sec:='biadbdaiibda'
    else if (tablero[2,3]=4) then sec:='bdaibiadbdaiibda';
    secuencia(sec);
  end;
  if hecho then exit;
  {ultimo paso}
  sec:='';
  if tablero[3,2]=6 then
    sec:='bdaiibdd';
  if tablero[3,2]=7 then
    sec:='ibdd';
  if tablero[3,2]=8 then
    sec:='dbiaibdd';
  secuencia(sec);
end;

procedure tPuzzle.desordenar;
var i:integer;
begin
  For i:=1 to 1000 do
    case random(4) of
      0:arriba;
      1:abajo;
      2:derecha;
      3:izquierda;
    end;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  puzzle.dibuja(canvas);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  randomize;
  puzzle.init;
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var i,j:integer; temp:TTablero; iguales:boolean;
```

```
begin
  if timer1.Enabled then exit;
  if (x>=100)and(x<=250)and(y>=100)and(y<=250) then
  with Puzzle do begin
    j:=(x - 100)div 50+1;
    i:=(y - 100)div 50+1;
    temp:=tablero;
    if (i-1>0)and(tablero[i-1,j]=0) then
       abajo
    else
      if (i+1<4)and(tablero[i+1,j]=0) then
         arriba
      else
        if (j-1>0)and(tablero[i,j-1]=0) then
          derecha
        else
          if (j+1<4)and(tablero[i,j+1]=0) then
            cambia(tablero[i,j+1],tablero[i,j]);
    iguales:=true;
    for i:=1 to 3 do
      for j:=1 to 3 do
        if not(temp[i,j]=tablero[i,j]) then
          iguales:=false;
    if not iguales then
      inc(n);
    label1.Caption:='Número de movimientos: '+IntToStr(n);
    dibuja(canvas);
    Button1.Enabled:=true;
    solucion:='';
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  n:=0;
  with puzzle do begin
    tableroinicial:=tablero;
    movimientos:=0;
    solucion:='';
    resolver;
    movimientos:=0;
    tablero:=tableroinicial;
    t:=solucion;
    hecho:=false;
    dibuja(canvas);
    timer1.Enabled:=true;
    Button1.Enabled:=false;
```

```
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  n:=0;
  with puzzle do begin
    desordenar;
    tableroinicial:=tablero;
    dibuja(canvas);
    movimientos:=0;
    solucion:='';
    Button1.Enabled:=true;
    timer1.Enabled:=false;
  end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  with puzzle do
    if hecho then begin
      timer1.Enabled:=false;
      Button1.Enabled:=true;
    end
    else begin
      inc(movimientos);
      secuencia(t[movimientos]);
      if not hecho then
        label1.Caption:='Número de movimientos:
'+IntToStr(movimientos)
      else
        Button1.Enabled:=true;
      dibuja(canvas);
    end
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  Application.Terminate;
end;

end.
```

Codigo del juego de "Mente Maestra"

```
uses crt;

const MaxColores = 6;
      col:string[6] = 'BCGRYW';

type
    TMasterMind = object
    private
      NumJugadas,paso,numCorrectas,numBien:integer;
      resuelto:boolean;
      Intento:array[1..30]of string[4];
      Objetivo:string[4];
    public
      procedure Init;
      procedure LeerObjetivo;
      procedure muestraJugada(n:integer);
      procedure leerMarcador;
      procedure juega;
      procedure MuestraResultado;
    end;

procedure TMasterMind.LeerObjetivo;
var ch:char; i:integer;
begin
  write('introduzca secuencia: ');
  objetivo:='';
  for i:=1 to 4 do begin
    repeat
      ch:=col[random(6)+1];
{      ch:=readkey;}
    until upcase(ch) in ['B','C','G','R','Y','W'];
    write(upcase(ch));
    objetivo:=objetivo+upcase(ch);
  end;
  writeln;
end;

procedure TMasterMind.MuestraResultado;
begin
  if objetivo=intento[paso] then
    writeln('resuelto en ',paso,' pasos.')
  else
    writeln('NO se encontro solucion en ',paso,' pasos.');
end;
```

```
procedure TMasterMind.Juega;
var
  i,j,k,l,m,p,total:integer;
  ficha:string[4];
  probada:array[1..24]of string[4];
  YaProbada:boolean;
begin
  total:=0;
  ficha:='    ';
  writeln('--- Primera etapa---');
  repeat
    inc(paso);
    Intento[paso]:=col[paso]+col[paso]+col[paso]+col[paso];
    muestraJugada(paso);
    leerMarcador;
    if numCorrectas>0 then
      for j:=1 to numCorrectas do begin
        inc(total);
        ficha[total]:=Intento[paso][j];
        if total = 4 then break;
      end;
  until (paso=MaxColores)or(total=4);
  writeln('--- Segunda etapa---');
  p:=0;
  for i:=1 to 24 do
    Probada[i]:='';
  for i:=1 to 4 do
    for j:=1 to 4 do
      if i<>j then
        for k:=1 to 4 do
          if (k<>i)and(k<>j) then begin
            l:=10-(i+j+k);
            {busca la combinaci$n para no repetirla}
            yaProbada:=false;
            for m:=1 to 24 do
              if (ficha[i]+ficha[j]+ficha[k]+
                 ficha[l])=probada[m] then
                yaProbada:=true;
            if not YaProbada then begin
              inc(paso);
              Intento[paso,1]:=ficha[i];
              Intento[paso,2]:=ficha[j];
              Intento[paso,3]:=ficha[k];
              Intento[paso,4]:=ficha[l];
              inc(p);
              probada[p]:=intento[paso];
```

```
              muestraJugada(paso);
              leerMarcador;
              if numCorrectas = 4 then exit;
            end;
          end;
end;

procedure TMasterMind.Init;
var i,j:integer;
begin
  for i:=1 to 30 do
    Intento[i]:='BBBB';
  paso:=0;
end;
procedure TMasterMind.muestraJugada(n:integer);
begin
  write(Intento[n]);
end;
procedure TMasterMind.leerMarcador;
var ch:char;
begin
  write('    Blancas =>>');
  repeat
    ch:=readkey;
  until ch in ['0'..'4'];
  numBien:=ord(ch)-ord('0');
  write(ch,' Negras =>>');
  repeat
    ch:=readkey;
  until ch in ['0'..'4'];
  write(ch,' ');
  numCorrectas:=ord(ch)-ord('0');
  writeln;
end;

var m:TMasterMind;

begin
  clrscr;
  with m do begin
    init;
    LeerObjetivo;
    juega;
    MuestraResultado;
  end;
  readln;
end.
```