Apunte Practico CBD

Bautista Garcia

April 28, 2024

Contents

1. Algoritmica Clasica

Declaracion de Archivos

```Pascal
RegistroHogares = Record
  Codigo_provincia: integer;
  Codigo_localidad: integer;
  Barrio: integer;
  Cantidad: integer;
end;
 tArchivo = File of RegistroHogares
```

Escritura en Archivos de Texto

```Pascal
writeln(reg.integer, reg.real, reg.string)
writeln(reg.string)
```

Para poder **reusar** los archivos de texto, se debe escribir un string por cada linea.

Mapeo Fisico - Logico + Posicionarse en Primer posicion

```Pascal
Assign(archivo, 'hogares.dd');
Reset(archivo);
// Archivo es la variable de tipo tArchivo
```

Lecturas

```Pascal
procedure leer(var archivo: detalle; var dato: venta_prod);
begin
 if (not(EOF(archivo))) then
  read (archivo, dato)
 else
  dato.cod := valoralto; //Se le pone un valor alto para chequear en CORTE FUTURO
end;
```

1.1. Corte de Control

```Pascal
 Leer(archivo, reg);
 While (reg .codProv <> valorAlto) do begin
   codProvAct:=reg.codProv;
   votosProvincia:=0;
   writeLn('Provincia ', codProvAct);
   while (reg.codProv=codProvAct) do begin {corta la ejecución cuando cambia pcia}
     codLocAct:= reg.codLoc;
     hogLocalidad:=0;
     write(' Localidad ', codLocAct);
     while (reg.codProv=codProvAct) and (reg. codLoc=codLocAct) do begin
       hogLocalidad := hogLocalidad + reg.cantidad;
       Leer(archivo, reg)
     end;
     hogProvincia := hogProvincia + hogLocalidad ;
```

```
15      writeLn('Hogares Plan Help localidad: ', hogLocalidad );
16   end;
17  writeLn('Hogares Plan Help Pcia: ', hogProvincia )
18  end;
19  Close(archivo);
```

El objetivo es presentar la informacion de un archivo de forma ordenada.

1. Recordar siempre **cerrar el archivo** al finalizar operacion (Linea 19).
2. En las condiciones del corte de control se van **sumando las condiciones**. *En este ejemplo puede ocurrir que aparezca una ciudad con el mismo nombre pero de **diferente provincia**, es por esto que anidamos las condiciones.*

1.2. Actualizacion Maestro - Detalle

```pascal
1   begin {programa principal}
2    assign(mae, 'maestro');
3    assign(det, 'detalle');
4    reset(mae);
5    reset(det);
6    regm.cod=valoralto;
7    leer(det, regd);
8    while (regd.cod <> valoralto) do begin {Se procesan todos los registros del
9   archivo detalle}
10     aux := regd.cod;
11     total := 0;
12     while (aux = regd.cod) do begin {suma el total vendido para = producto}
13       total := total + regd.cant_vendida;
14       leer(det, regd);
15   end;
16    while (regm.cod <> aux) do {se busca el producto detalle en el maestro}
17       read (mae, regm);
18    regm.cant := regm.cant - total;
19    seek(mae, filepos(mae)-1);
20    write(mae, regm); {se actualiza el maestro}
21    if (not(EOF(mae))) then read(mae, regm);
22   end;
23  close(det);
24  close(mae);
25  end
```

1.3. Merge

Procedimiento de Minimos

```pascal
1   procedure minimo(var det: adet; var rdet:ardet;var min:producto);
2   Var
3     posMin :int
4   Begin
5     posMin:=1;
```

```
6     min := rdet[posMin];
7     for i:=2 to DIML do begin
8       if(rdet[i].codigo < min.codigo)then begin
9         min:= rdet[i];
10        posMin:=i; end;
11      end;
12    leer(det[posMin],rdet[posMin]);
13  end;
```

Guardamos en **min** el minimo de todos los detalles.

1. for de **2 .. DIML** ya que inicializamos min con elemento de la primer posicion.
2. **Leer()** del final, para avanzar una posicion en el detalle donde encontramos minimo, para que en el proximo minimo() no se vuelva a evaluar el minimo anterior.

```pascal
1   begin
2     // Logico <-> Fisico + Creacion
3     assign (mae, 'maestro');
4     rewrite (mae);
5
6     for i:=1 to 3 do begin
7       writeln('ESCRIBA UN NOMBRE PARA EL ARCHIVO: ');
8       read(nombreDet);
9       assign(det[i],nombreDet);
10    end;
11    // Leemos detalles y guardamos en arreglo de registro detalle
12    for i:=1 to 3 do begin
13      reset (det[i]);
14      leer (det[i], regd[i]);
15    end
16    minimo (det, rdet, min);
17
18    while (min.codigo <> valoralto) do begin
19      prod.codigo:= min.codigo;
20      prod.pu=min.pu;
21      prod.cant := 0;
22      while (min.codigo = prod.codigo) do
23        prod.cant := prod.cant + min.cant;
24        minimo (det, rdet, min);
25      end;
26      write (mae, prod);
27    end;
28    close(mae);
29    for i:=1 to 3 do close (det[i])
30  End.
```

2. Bajas

2.1. Baja Fisica

```pascal
Procedure BajaFisica(var a:archivo; apellido,nombre: String);
Var
  posBorrar:integer; rp,aux:per; apellido, nombre: st20;
begin
  reset(a);
  readln(apellido); readln(nombre);
  rp.ape= ‘zzz’;  rp.nom=‘zzz’;
  // Buscamos NOMBRE y APELLIDO
  while (not eof(a) and ((rp.ape != apellido) and (rp.nom!=nombre))) then
    read(a,rp);
  // Chequeamos si encontramos o si corto por EOF
  if(rp.ape = apellido) and (rp.nom=nombre)then begin
    posBorrar:=filepos(a)-1;
    // Escribimos ultimo reg en posicion de borrado
    seek(a,filesize(a)-1);
    read(a,aux);
    seek(a, posBorrar);
    write(a,aux);
    // Nos paramos en registro movido y ponemos nueva marca de EOF con truncate
    seek(a,filesize(a)-1);
    truncate(a);
  end,
  close(a);
end;
```

2.2. Baja Logica

```pascal
Procedure BajaLogica(var a: archivo; apellido,nombre: String);
Var
  posBorrar:integer; rp:persona; apellido, nombre: st20;
begin
  reset(a);
  readln(apellido); readln(nombre);
  rp.ape= ‘zzz’; rp.nom=‘zzz’;
  // Buscamos NOMBRE Y APELLIDO
  while (not eof(a) and ((rp.ape != apellido) and (rp.nom!=nombre))) then
    read(a,rp);
  If(rp.ape = apellido) and (rp.nom=nombre)then begin
    // Ponemos marca de borrado como # o @ en algun campo del registro
    posBorrar:=filepos(a)-1;
    rp.ape=“@”
    seek(a, posBorrar);
    write(a,rp);
  end;
  close(a);
end;
```

2.3. Lista Invertida

2.3.1. Altas

```pascal
begin
  Reset(a); Read(a, sLibre); {lee la cabecera}
  Val(sLibre, nLibre,cod); {convierte de string a number}
  // Si no hay espacio libre nos paramos en EOF
  If (nLibre= -1) then
    Seek(a, FileSize(a))
  else begin
    // Cabecera pasa a apuntar al segundo lugar libre
    Seek(a, nLibre);
    Read(a, sLibre); {lee la posición a reutilizar}
    Seek(a, 0);
    Write(a, sLibre); {Actualiza la cabecera}
    // Nos posicionamos en lugar libre nuevamente
    Seek(a, nLibre);
  end;
  // Escribimos, ya sea en EOF o en pos libre
  WriteLn('Raza Vacuna: ');
  ReadLn(r); {Lee la raza de teclado}
  Write(a, r); {Guarda la raza}
  Close(a)
end;
```

2.3.2. Bajas

```pascal
begin
  Reset(a);
  Write('Nombre de la raza a dar de baja: ');
  ReadLn(rb); {Raza eliminar}
  Read(a, sLibre) {lee la cabecera}
  r= ‘zzz’;
  While (not((r=rb) or EoF(a))) Read(a, r);
  // Chequeamos que no sea EOF
  If r=rb then begin
     // Ahora el nuevo espacio libre apunta al espacio libre que apuntaba la cabecera
    nLibre:=FilePos(a)-1;
    Seek(a, nLibre);
    Write(a, sLibre); //COPIAMOS PTR DE CABECERA A ESP LIBRE

    // Actualizamos la CABECERA, para que apunte al nuevo espacio libre
    Str(nLibre, sLibre); {Convierte de number a string}
    Seek(a, 0);
    Write(a, sLibre); {Se actualiza la cabecera}
  end
  else Begin {no se encuentra la raza}
    writeln("No se encontro nada")
  Close(a)
end
```

3. Arboles B

Algunas diferencias con la teoria en esta clase de arboles son:
- Estos arboles deben tener un minimo de $\left[\frac{M}{2}\right] - 1$ elementos en todos los nodos, ya sean nodos hojas o nodos interiores.

3.1. Altas en B

Buscamos donde se debe insertar el nodo recorriendo el arbol.
1. Si hay **espacio libre** lo insertamos y se termina la operacion de alta.
2. Si no hay espacio libre hacemos el tratamiento de **overflow**.

Overflow: Sobrepasamos la cantidad maxima de elementos en un nodo

1. Si el overflow ocurre en la **raiz** creamos dos hijos (nodos nuevos o alguno que habia quedado liberado) y si es en un nodo $\neq$ raiz creamos un hermano derecho (nodo nuevo o alguno que habia quedado liberado) y promocionamos el **separador** al padre. Luego para los conjuntos a)$\{20, 23, 55, 60, 70\}$ b)$\{20, 23, 24, 55, 60, 70\}$ la redistribucion es la siguiente.

$$a)\ [20, 23]_{\text{hijo izquierdo}}\ [55]_{\text{padre}}[60, 70]_{\text{hijo derecho}} \tag{1}$$

$$b)\ [20, 23, 24]_{\text{hijo izquierdo}}\ [55]_{\text{padre}}[60, 70]_{\text{hijo derecho}} \tag{2}$$

En arboles B, siempre le mandamos mas al hijo izquierdo en caso de no quedar cantidades equitativas.

2. Si al mandar un elemento al padre, hay espacio libre, se coloca este **separador** y termina la operacion. Si no hay espacio libre se produce un **nuevo overflow** y repetimos 1) con este nodo.

Costos de Operacion:
- Cuando buscamos en que nodo insertar al nuevo elemento, vamos a tener que **leer** cada nodo incluyendo al nodo donde corresponde el elemento para ver si entra alli o si va a haber overflow.

1. Si entra en el nodo correspondiente vamos a tener una unica **escritura** y todas las lecturas hasta llegar a ese nodo.
2. En caso de overflow vamos a escribir en el hijo izquierdo, luego en el hijo derecho (crear + escribir = 1 escritura) y luego en el padre (si hay espacio, sino otro overflow). Es por esto que un overflow (con espacio en padre) resulta en *3 escrituras.*

Orden de creacion de nodos y operaciones L/E es izquierda, derecha, arriba.

3.2. Bajas en B

Buscamos el elemento a eliminar en el arbol y luego si es un nodo interno se cambia por su **menor de sus claves derechas** para luego operar:
1. No se encuentra el elemento buscado.
2. Si al eliminar se sigue cumpliendo la propiedad de **minima cantidad**, finaliza la operacion de baja.

3. Si al eliminar no se cumple la propiedad de minimos, se debe hacer un tratamiento de **underflow**

Underflow: Cantidad de elementos luego de borrar resulta menor a $\left[\frac{M}{2}\right] - 1$.

1. Siempre que se pueda se va a priorizar tratar underflow con **redistribucion** por sobre **concatenacion**. *Si podemos elegir la politica el orden de prioridad es 1) Redistribucion por izquierda/derecha 2)Redistribucion por izquierda/derecha 3)Concatenacion por izquierda/derecha.* En caso de no poder elegir el orden es *1)Redistribucion por politica indicada 2)Concatenacion por politica indicada.*

Redistribucion: *Esto se puede hacer si la* $\frac{\text{cantidad de elementos}}{2}$ *es mayor al minimo de cada nodo* . Tomamos todos los elementos de nodos hermanos adyacentes elegidos y el elemento separador. Conjunto a) $\{20, 23, 55, 60, 70\}$ con 55 (elemento separador del padre).

$$[20, 23]_{\text{nodo izq}}[55]_{\text{padre}}[60, 70]_{\text{nodo der}} \tag{3}$$

Fusion o Concatencacion: *Alternativa en caso de no poder redistribuir* . Tomamos todos los elementos de nodos hermanos adyacentes elegidos y el elemento separador. Conjunto a) $\{20, 23, 55\}$ con 23 (elemento separador del padre).

$$[20, 23, 55]_{\text{nodo izq}} \tag{4}$$

Ademas la clave 23 (elemento separador) se borra del padre, se reacomodan los elementos restantes y el **nodo derecho se libera**.

Costos de Operacion:

- Cada vez que tomamos un elemento adyacente ademas de **escribir en el cuando redistribuimos**, previamente debemos **leerlo**. Ademas, cuando borramos el separador en el nodo padre, eso cuenta como una escritura.

4. Arboles B+

El **arbol b+** debe estar mas cargado a la derecha

4.1. Altas en B+

Se manejan igual que las altas en **B** con la diferencia que dado un conjunto a) $\{20, 23, 40, 50, 55\}$ donde ocurrio **overflow en un nodo hoja**:

$$[20, 23]_{\text{nodo izq}}[40, 50, 55]_{\text{nodo der}} \tag{5}$$

Ademas se debe enviar una copia al padre del menor de sus hijos derechos, en este caso 40. NOTA: Si el overflow es de un nodo interior, enviamos el elemento separador (no una copia) y la distribucion nos queda como en **altas en B**.

4.2. Bajas en B+

Muy similar a B con algunos detalles:

- Se prioriza la fusion a izquierda en caso de poder elegir.

- La clave de el elemento separador no se lleva al conjunto que vamos a redistribuir, unicamente se agarran los elementos de los nodos hermanos adyacentes. Lo que si, la clave del elemento separador se sigue borrando como haciamos en **arboles B**.

Redistribucion: Si la cantidad de nodos en ambos hermanos nos permiten **redistribuir**, borramos el elemento separador, mandamos cantidades equitativas a cada nodo y luego enviamos copia del **menor elemento del nodo derecho** como **separador** al padre.

Fusion: Al fusionar, siempre fusionamos a **izquierda** y luego borramos el elemento separador de ambos nodos que fusionamos (lo que nos puede ocasionar otro **underflow**).

NOTAS:
- Al reemplazar claves de nodos internos por elementos de sus hojas (para ser eliminados posteriormente). La convencion de la catedra es tomar el menor de sus hijos derechos.
- Si al **fusionar** queda un nodo libre, debemos escribir **LIBRE** N_i.

5. Dispersion

5.1. Saturacion Progresiva

5.2. Altas

Al ingresar en una tabla, calculo la direccion base, *"CLAVE"* mod *"NODOS"*:
1. Si hay espacio libre lo inserto en alguno de sus registros.
2. Si no hay espacio libre comienzo una busqueda secuencial de espacio libre a partir de la direccion base y lo inserto.

5.3. Bajas

Si al borrar tengo a todos los registros hermanos ocupados, pongo marca de borrado para indicar que ese nodo estaba lleno (para futuras busquedas). Sino, lo borro y se termina la operacion.

5.4. Saturacion Progresiva Encadenada

5.5. Altas

Calculo direccion base:
1. Si hay espacio libre la inserto.
2. Si no hay espacio libre, pero en ese lugar hay una **clave intrusa** (clave insertada por saturacion en proximo espacio libre). Entonces muevo la clave intrusa al proximo espacio libre, reacomodo el puntero que la apuntaba y por ultimo inserto la nueva clave en su **direccion base correspondiente**.
3. Si no hay espacio libre y la clave que ocupa ese lugar es correcta, busco el proximo espacio libre inserto la clave nueva (como intrusa) y apunto el puntero de la ocupada a donde inserte la nueva clave.

NOTA: Si ocurre 3, y ya hay una "lista de encadenadas", entonces agrego el nuevo elemento entre el primero y el segundo. Ejemplo: Alta de 23

$$\begin{gathered} 78 \to 56(-1) \\ 78 \to 23 \to 56(-1) \end{gathered} \tag{6}$$

El 78 y 56 no cambian de lugar, solo se reacomodan sus punteros. El 23 va a ir a la proxima direccion libre

Pregunta: La proxima direccion libre se busca a partir del ultimo elemento de la lista de encadenados??

5.6. Bajas

1. Si al borrar, el elemento era nodo intermedio de la lista de encadenados, borramos esa elemento de la tabla y reacomodamos punteros. Ejemplo: –23.

$$\begin{gathered} 78 \to 23 \to 56(-1) \\ 78 \to 56(-1) \end{gathered} \tag{7}$$

2. Si al borrar, el elemento era el primer nodo de la lista de encadenados, borramos ese elemento y traemos al segundo de la lista a donde estaba ese elemento. Luego el ultimo elemento de la lista, es la posicion de la tabla que quedara libre.

NOTA: En bajas de SPE no usamos **marca de borrado**.

5.7. Saturacion Progresiva Encadenada con area separada

En este tratemiento de saturacion, si no hay mas espacio libre en el area principal, enviamos al area separada. Luego alli se comportan igual que la **Saturacion progresiva encadenada**.

5.8. Dispersion Doble

5.9. Altas

Aplico 1er funcion de dispersion F_1:

1. Hay espacio libre $\to$ inserto.
2. No hay espacio libre:

Aplico 2da funcion de dispersion F_2:

$$D = F_1 + F_2 + 1 \tag{8}$$

1. Si D esta libre inserto, si no esta libre:

$$D_{\text{nuevo}} = D_{\text{anterior}} + F_2 + 1 \tag{9}$$

Asi hasta encontrar un lugar libre.

5.10. Bajas

Buscamos el elemento con el mismo algoritmo que usamos para las altas. En **Dispersion Doble** todas las bajas llevan **marca de borrado**.

Pregunta: Para que sirven las marcas de borrado y porque se ponen en algunas bajas en Saturacion progresiva, y en todas las bajas de Dispersion Doble.

5.11. Dispersion Extensible

Tengo una tabla de hash, que para cada clave (de cualquier tipo) me retorna una secuencia **N bits**. Luego en cada direccion de disco (nodo de disco) voy a tener capacidad de almacenar **K registros**. A su vez cada nodo de disco va a tener un **valor asociado** (**VA**) (arrancan en 0).

VALOR ASOCIADO: Con cuantos bits (contando desde el menos significativo) vas a direccionar a ese nodo.

1. Voy a insertar los registros hasta **saturar el nodo**. Cuando se satura incremento su **VA** en 1 y creo un nodo nuevo con el mismo **VA** que el *nodo saturado*.

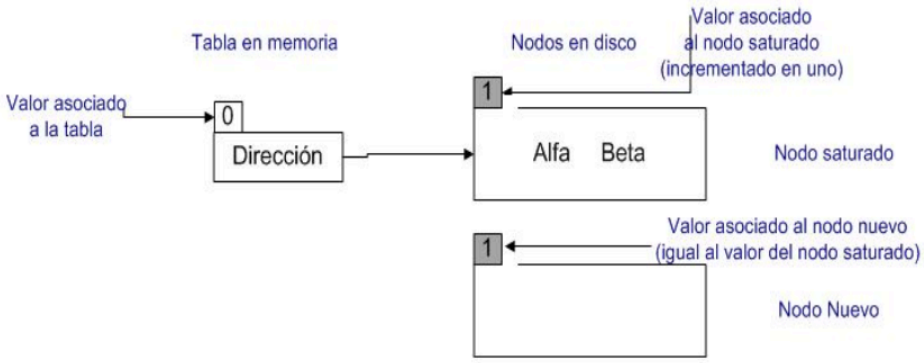

Figure 1: Gamma satura al nodo donde estaba Alfa y Beta

*Si el **VA** de un nodo es **mayor estricto** que el **VA** de la tabla, se debe **duplicar los espacios de la tabla** e **incrementar su VA en 1**.*

2. Para acomodar *Beta, Gamma y Alfa* debemos ver los bits de la **hash table**. Vamos a usar tantos bits como su **VA**, contando desde el menos significativo:

Clave	FH(clave)
Alfa	00.....1001
Beta	00.....0100
Gamma	00.....0010

Figure 2: En este caso $\mathrm{VA} = 1 \therefore$ usamos 1(Alpha), 0(Betta y Gamma)

Nos queda la asociacion

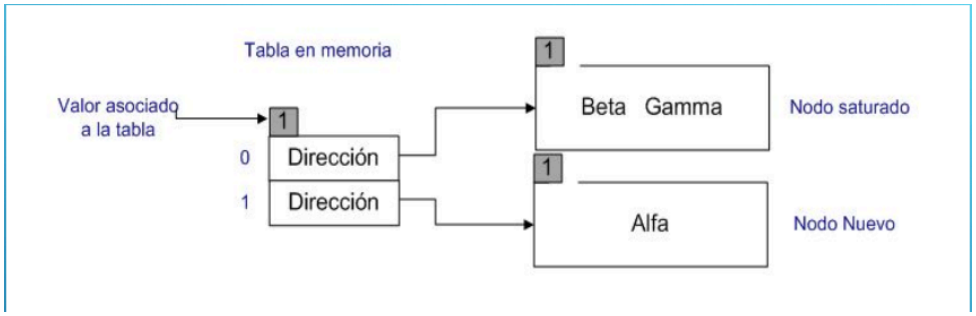

Figure 3: Incrementamos VA de tabla, y duplicamos espacio respecto al primer estado

3. Cuando llega **epsilon** este debe ir a donde esta Beta y Gamma (nos fijamos en HT) y vemos que el nodo esta lleno. Es por eso que repetimos ***Paso 1***. A su vez como tienen VA = 2, al

conjunto {Beta, Gamma, Epsilon} lo vamos a direccionar usando 2 bits menos significativos de la tabla, mientras que al nodo {Alfa, Delta} lo vamos a seguir direccionando con el bit menos significativo (es por eso que lo apuntan 2 direcciones).

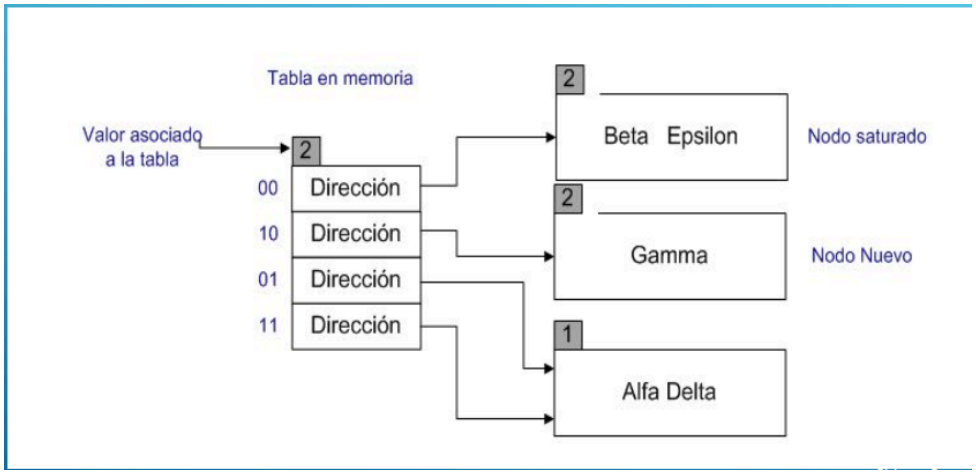

Figure 4:

4. Cuando llega Rho seguimos con los mismos procedimientos, pero al hacer **paso 1**, observamos que **VA nodo** no supera **VA tabla** ∴ no duplicamos la tabla de memoria (a su vez a todos los que tienen **VA = 2** los direccionamos con 2).

Bibliography

Index of Figures