Unidad 3: Diseño y Análisis de Algoritmos

Tema VI: Ordenación y Búsqueda. Introducción a la ordenación. Clasificación de los algoritmos de ordenación. Métodos Directos. Métodos Logarítmicos. Intercalación. Ordenación por montículos. Problema de la Búsqueda Estática. Búsqueda Secuencial. Búsqueda Binaria. Búsqueda Interpolada. Cola de prioridades.

Mgter. Oscar Adolfo Vallejos
FaCENA - UNNE

PARTE PRIMERA

Búsqueda, Clasificación e Intercalación

Búsqueda

- La operación de búsqueda nos permite encontrar datos que están previamente almacenados. La operación puede ser un éxito, si se localiza el elemento buscado o un fracaso en otros casos.

- La búsqueda se puede realizar sobre un conjunto de datos ordenados, lo cual hace la tarea más fácil y consume menos tiempo; o se puede realizar sobre elementos desordenados, tarea más laboriosa y de mayor insumo de tiempo.

Búsqueda

- *Búsqueda Interna* será aquella acción que se realice sobre datos que se encuentran en la memoria principal, por ejemplo en un arreglo.

- *Búsqueda Externa* es cuando todos sus elementos se encuentran en memoria secundaria (archivos almacenados en dispositivos de cinta, disco, etc.-)

- La operación de búsqueda de un elemento X en un conjunto consiste en determinar si el elemento X pertenece al conjunto y en este caso dar su posición, o bien, determinar que el elemento X no pertenece al conjunto.

- Los métodos más usuales para la búsqueda son:

- *Búsqueda secuencial o lineal*

- *Búsqueda binaria*

- *Búsqueda por transformación de claves*

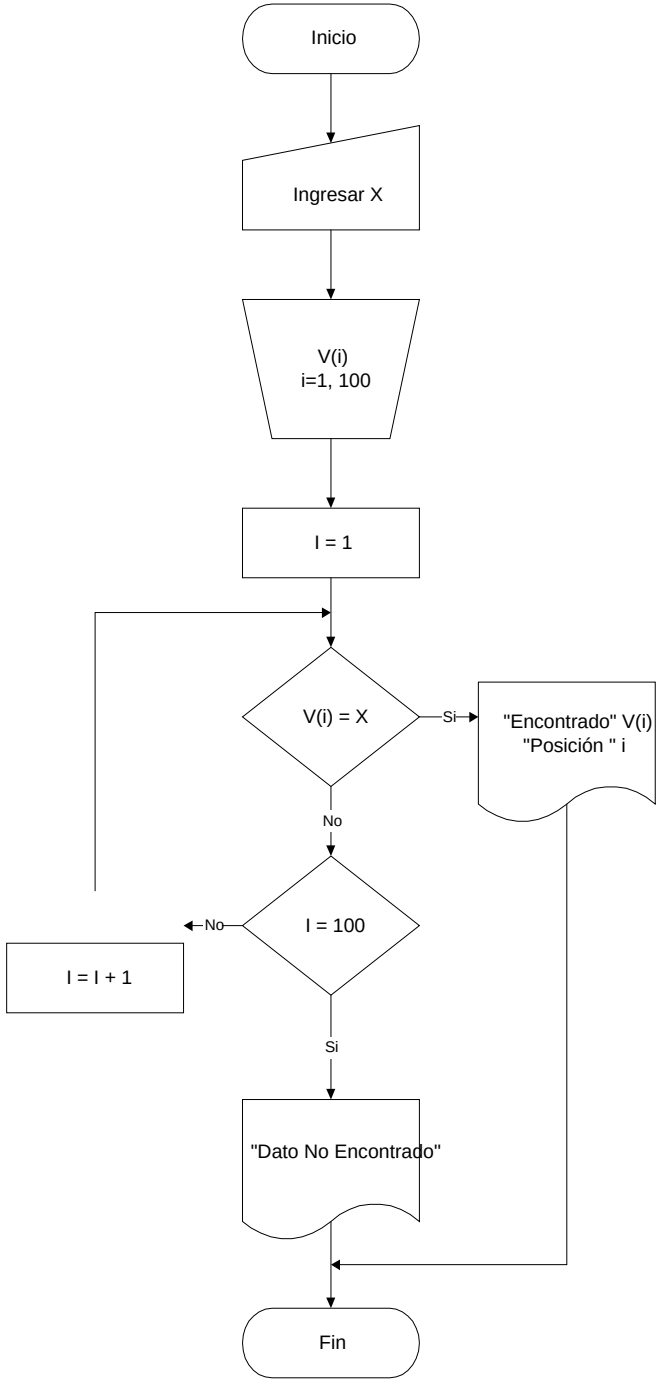
Búsqueda Secuencial

- La búsqueda secuencial es la técnica más simple para buscar en una lista de datos. Este método consiste en recorrer una lista (o arreglo) en forma secuencial y comparar cada elemento del arreglo (o de la lista) con el valor deseado, hasta que éste se encuentre o finalice el arreglo (o la lista).

- Normalmente cuando la función de búsqueda termina con éxito, es decir encontró el dato buscado, interesa conocer en que posición fue encontrado el dato buscado. Esta idea se puede generalizar en todos los métodos de búsqueda.

- La búsqueda secuencial no requiere ningún requisito para el arreglo, y por lo tanto no necesita estar ordenado.

Inicio
Ingresar X
Leer V(100)
Desde I = 1 hasta 100 **hacer**
Si V(i) = X **entonces**
Imprimir “
Encontrado” V(i), “Posición” i
Fin
Fin Si
Fin desde
Imprimir “Dato no encontrado”
Fin



Código en C: Búsqueda del mayor elemento en un vector

```
#include <stdio.h>
void main(){

int vector[10];
int i;
int mayor=0;

for (i=0; i<10; i++){
  printf( "ingrese 10 numeros enteros. Numero %d\n", i+1);
  scanf("%d", &vector[i]);
}

for (i=0; i<10; i++){
   if (vector[i]> mayor){
   mayor=vector[i];
   }
}
printf("El mayor es %d\n", mayor);
}
```

Búsqueda Secuencial

Consideraciones

- El método es solo adecuado para listas cortas de datos.
- A la hora de analizar la complejidad del método secuencial, tenemos que tener en cuenta el caso mas favorable y el mas desfavorable.
- Cuando el elemento no se encuentra tiene que realizar las *n* comparaciones. Y en los casos en que el elemento buscado se localiza, este podrá estar en el primer lugar, en el último o en un lugar intermedio.
- Entonces, al buscar un elemento en un arreglo de N componentes se harán:
- *N comparaciones* si el elemento no se localiza
- *N comparaciones* si el elemento está en la última posición
- *1 comparación* si está en el primer lugar
- *i comparaciones* si está en un lugar intermedio (posición 1 < i < N)
- Analizando lo escrito anteriormente podemos suponer que el número medio de comparaciones a realizar es de ***(n + 1) / 2*** , que es aproximadamente igual a la mitad de los elementos de la lista.

Búsqueda Secuencial

- Una aplicación de la búsqueda puede ser en:
- Vector: Se recorre el vector de izquierda a derecha hasta encontrar una componente cuyo valor coincida con el buscado ó hasta que se acabe el vector. En este último caso el algoritmo debe indicar la no-existencia de dicho valor. Este método es válido para vectores ordenados o no, aunque para los vectores ordenados veremos otros métodos mas eficientes.

- Vector Ordenado: Cuando el vector de búsqueda esta ordenado se consigue un algoritmo mas eficiente, sin mas que modificar la condición de fin del caso anterior. La ventaja que se obtiene es que una vez sobrepasado el valor buscado, no es necesario recorrer el resto del vector para saber que el valor no existe.

- Matriz: Se realiza mediante el anidamiento de dos bucles de búsqueda HASTA cuya finalización vendrá dada por la aparición del valor buscado o la terminación de la matriz. El recorrido se podrá hacer indistintamente por fila o por columna.

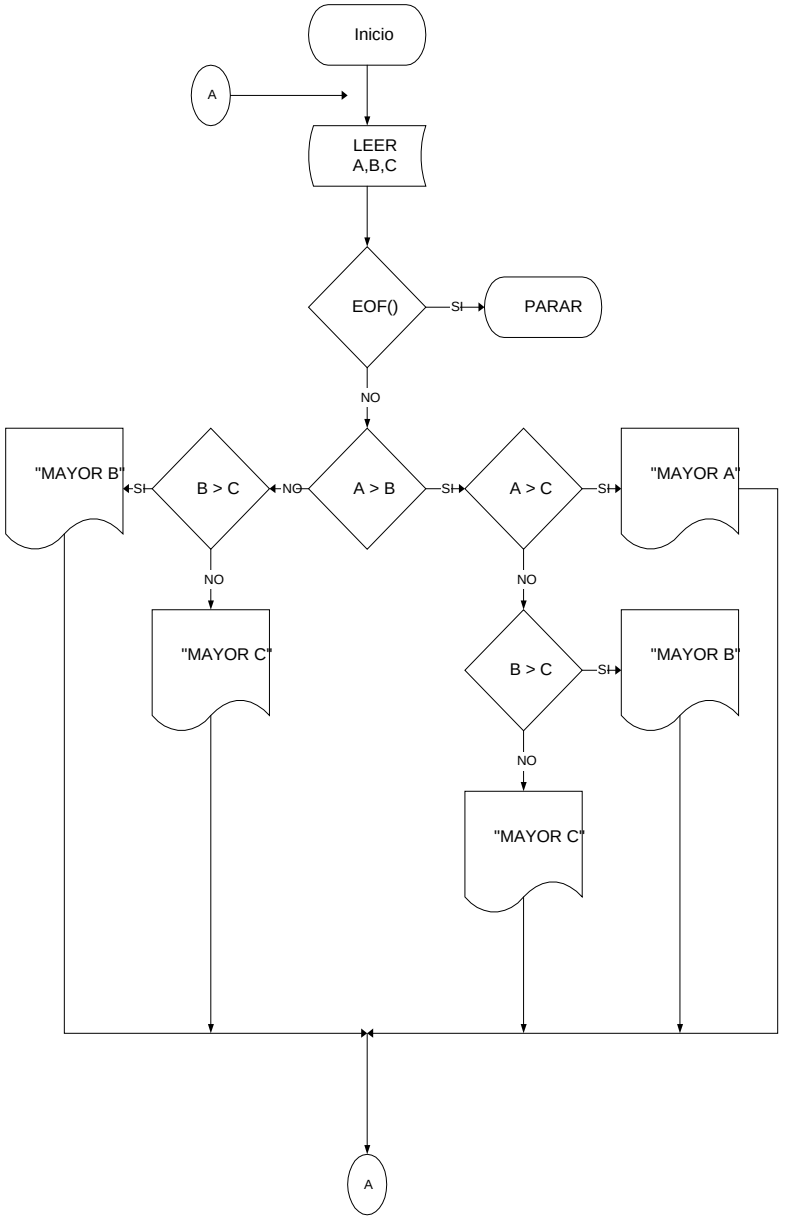
Búsqueda de máximos y mínimos

- En muchos casos, es necesario determinar el mayor o el menor valor de un conjunto de datos.
- Existen diversos algoritmos para esta determinación, en la mayoría de ellos se realizan comparaciones sucesivas de todos y cada uno de los datos resguardando en una variable auxiliar el valor que resulte mayor o menor, de acuerdo a lo que se busque, de manera tal que cuando no existan mas datos para comparar, esta variable auxiliar contendrá el valor máximo o mínimo buscado.

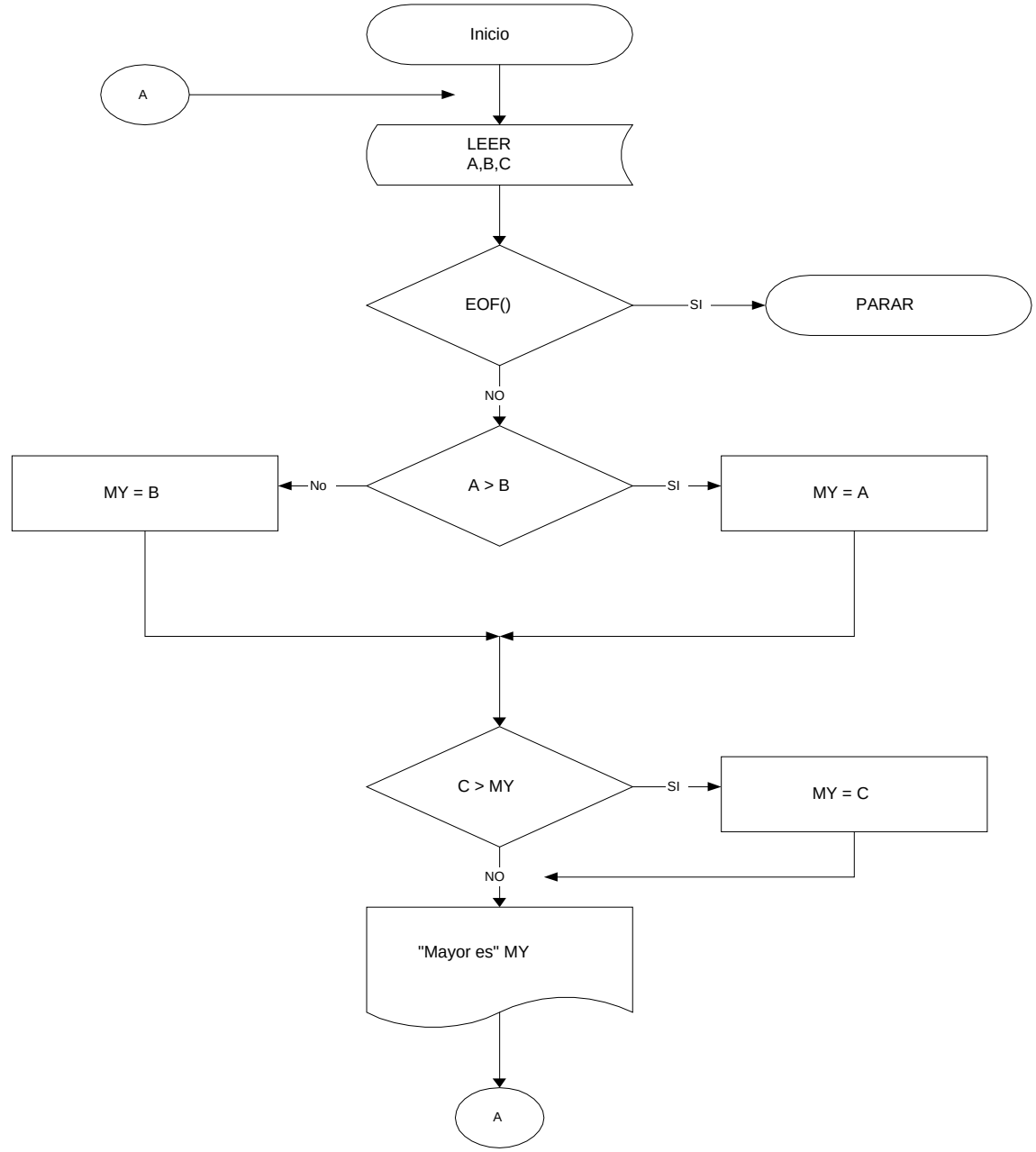
- Existen tres métodos para la resolución de este problema:
  - *Ramificación del árbol*
  - *Campeonato*
  - *Supuesto o prepo*

Ramificación del árbol: Consiste en las ***combinaciones de comparaciones*** de todas las variables que intervienen. Este método se realiza teniendo en cuenta que todos los campos deben estar simultáneamente en memoria (es del tipo de *búsqueda interna)*. Este método es al que tienden naturalmente todos los alumnos, en primera instancia, para la resolución del problema.

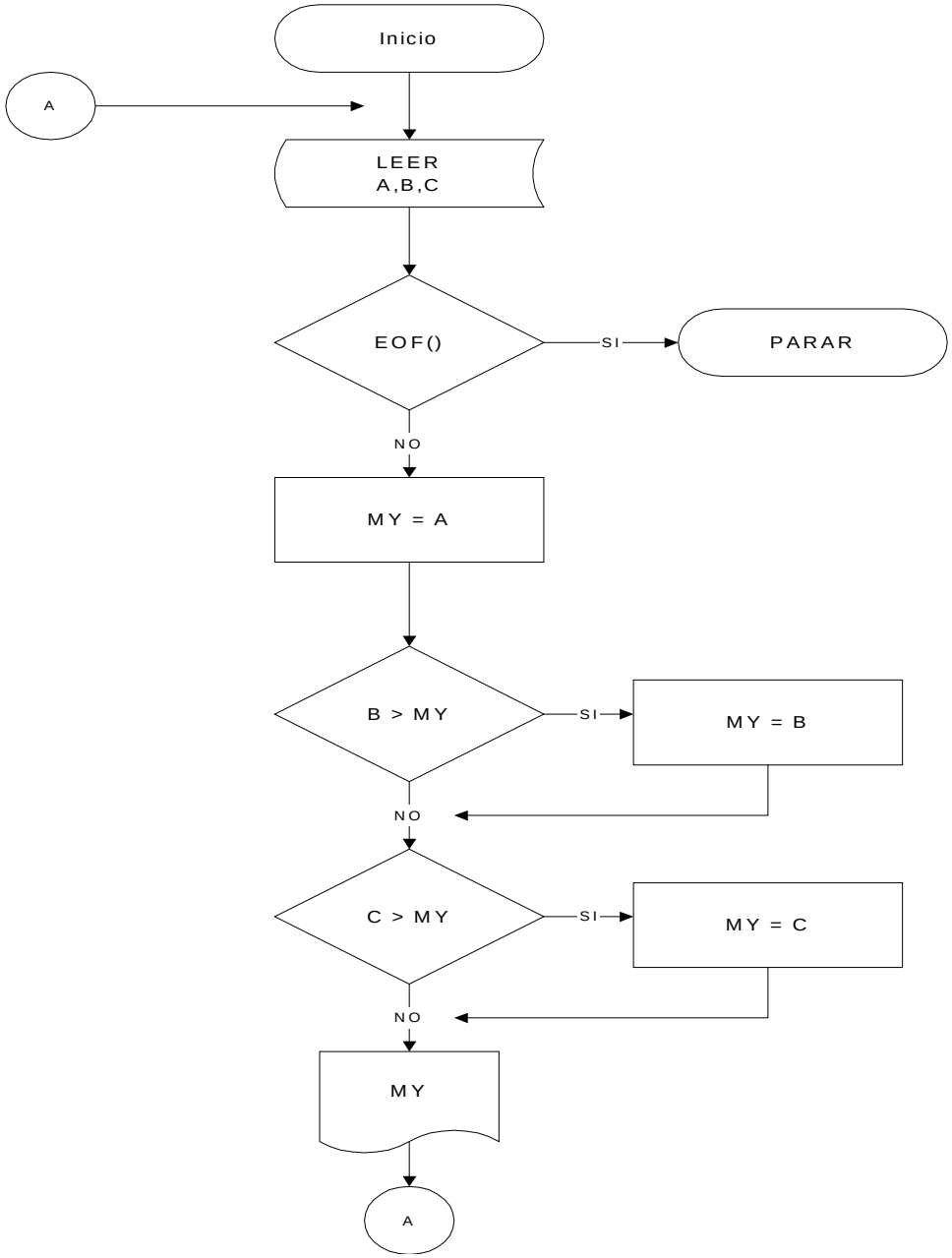
Comenzar
Mientras no EOF
Leer A,B,C
Si A>B
Entonces
Si A>C
Entonces
Imprimir “Mayor “A
Si_no
Si B>C
Entonces
Imprimir
“Mayor “B
Si_no
Imprimir
“Mayor “C
Fin_si
Fin_si
Si_no
Si B>C
Entonces
Imprimir “Mayor “B
Si_no
Imprimir “Mayor “C
Fin_si
Fin_si
Fin_mientras
Parar



- **<u>Campeonato</u>**: Consiste en la comparación **de a pares** de todas las variables que intervienen. En este método los campos también deben estar simultáneamente en memoria.



- **<u>Supuesto o Prepo</u>**: Es el que mas utilizaremos a lo largo del curso y consiste en suponer que una de las variables que existen en memoria, en el mismo momento, es mayor o menor de todas, y luego se realiza las comparaciones sucesivas con las restantes. Este método se adapta para los algoritmos de *búsqueda externa* (los campos no están simultáneamente en memoria, sino que ingresan registro a registro).



Código en “C”

```
void maximoMinimo (int *v, int tam, int *maximo, int *minimo){
    int i=0;
    //el primero valor del vector se lo ponemos a maximo y minimo
    //para ir comparando el resto de valores del vector
    *minimo=*maximo=v[i];

    for (i=0; i
     //si el siguiente valor del vector es mayor que el primero lo
    //guardamos en maximo y si es menos se guarda en minimo
    if (v[i]>*maximo)
    *maximo = v[i];
    if (v[i]<*minimo)
    *minimo = v[i];
    }
    }
```

búsqueda externa.

Debemos tener en cuenta para el desarrollo del método con búsqueda externa lo siguiente:

a) El ingreso de la información: los datos ingresan de a uno en la memoria.

b) Se supone al primer dato como mayor (no cualquiera de los campos como en búsqueda interna)

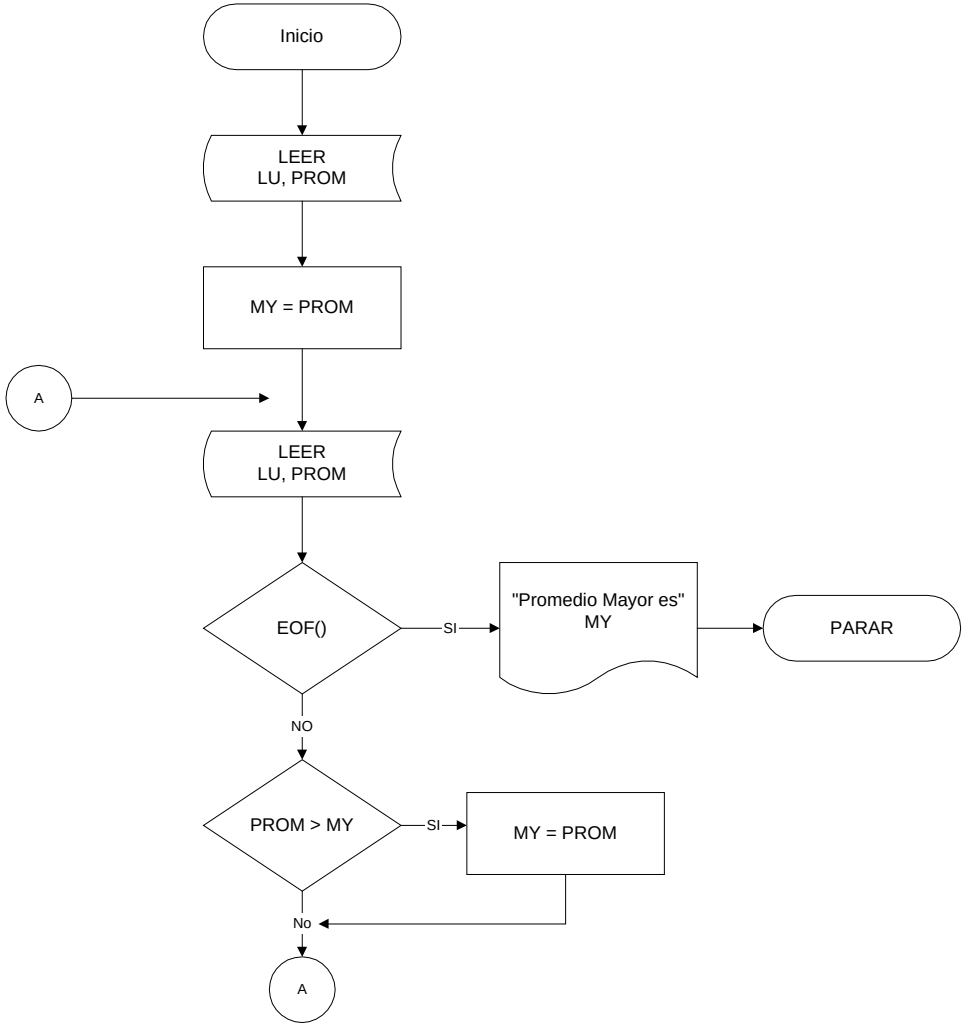
c) Se realiza la comparación de una variable (nota promedio) con respecto a la variable auxiliar MAYOR, y se repite dicha comparación hasta el fin del lote de datos, formando de esta manera un CICLO DE BUSQUEDA.

Las mismas consideraciones son válidas para la búsqueda de MINIMOS.

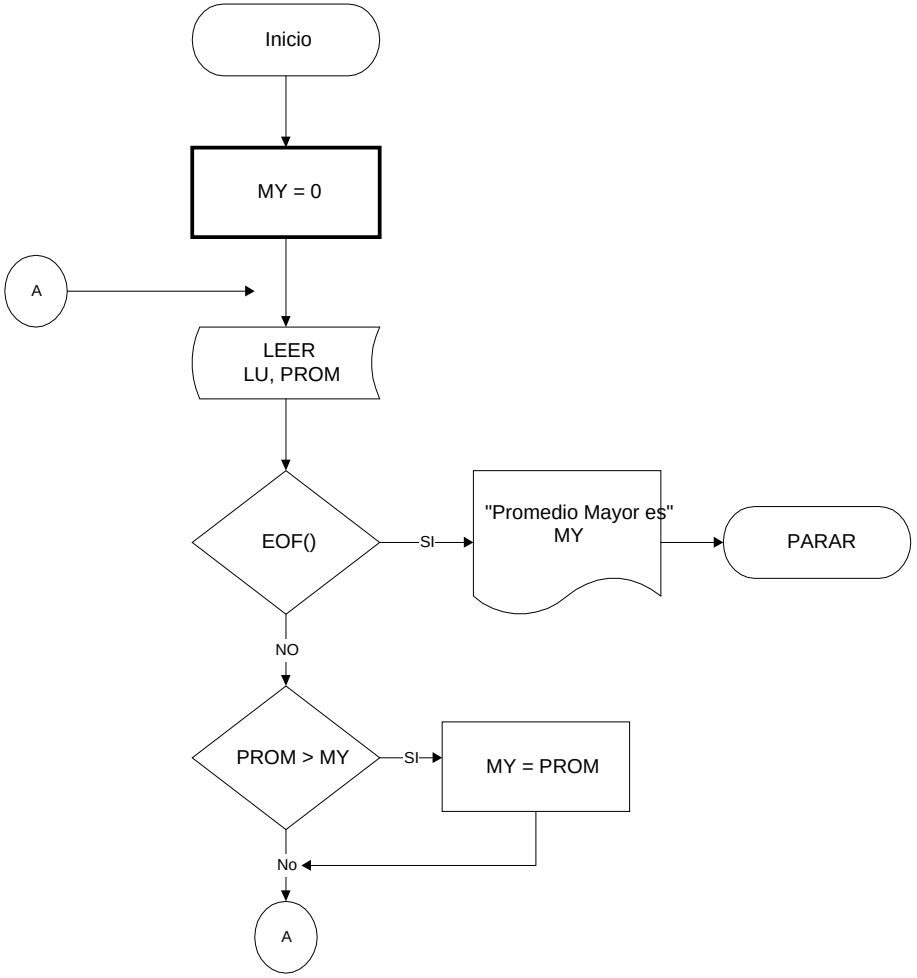
Una variante de este ejemplo es que la variable auxiliar tenga un valor inicial, que pueda darse de la siguiente manera:

•Para la determinación de máximos **se colocará en la variable el menor valor** que por la naturaleza de los datos pueda requerirse.

•Para la determinación del mínimo **se colocará en la variable el mayor valor** que esta pueda tomar, en función de la naturaleza de los datos.



Cabe aclarar que en el último ejemplo se evita la doble lectura.
El uso de estas técnicas, en búsquedas internas, es aconsejable cuando manejamos un número reducido de variables, con las salvedades y restricciones que en cada uno de los métodos ya mencionamos. En este caso se recomienda el uso de variables con subíndices o subindicadas.



Búsqueda binaria

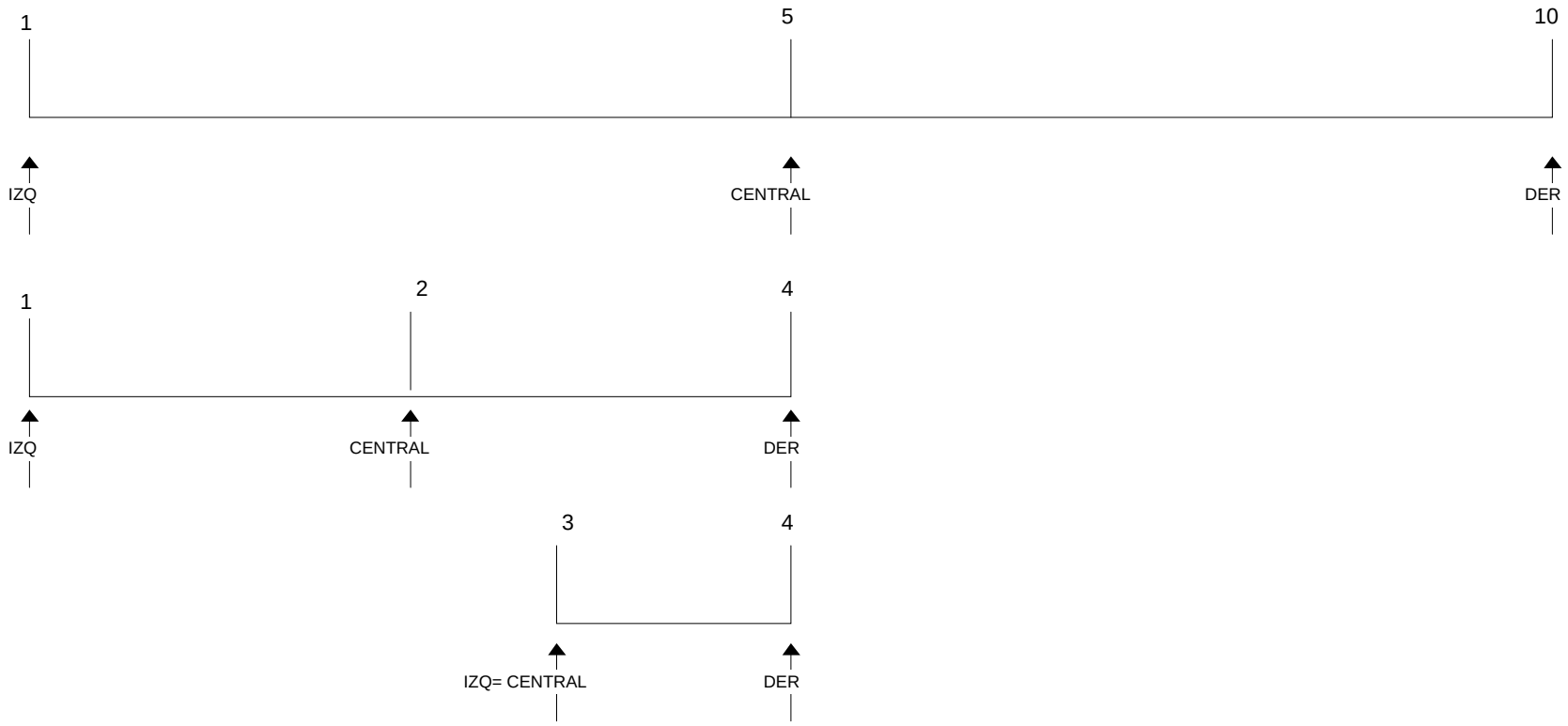
Es válido exclusivamente para datos ordenados y consiste en comparar en primer lugar con la componente central de la lista, y si no es igual al valor buscado se reduce el intervalo de búsqueda a la mitad derecha o izquierda según donde pueda encontrarse el valor a buscar.

El algoritmo termina si se encuentra el valor buscado o si el tamaño del intervalo de búsqueda queda anulado.

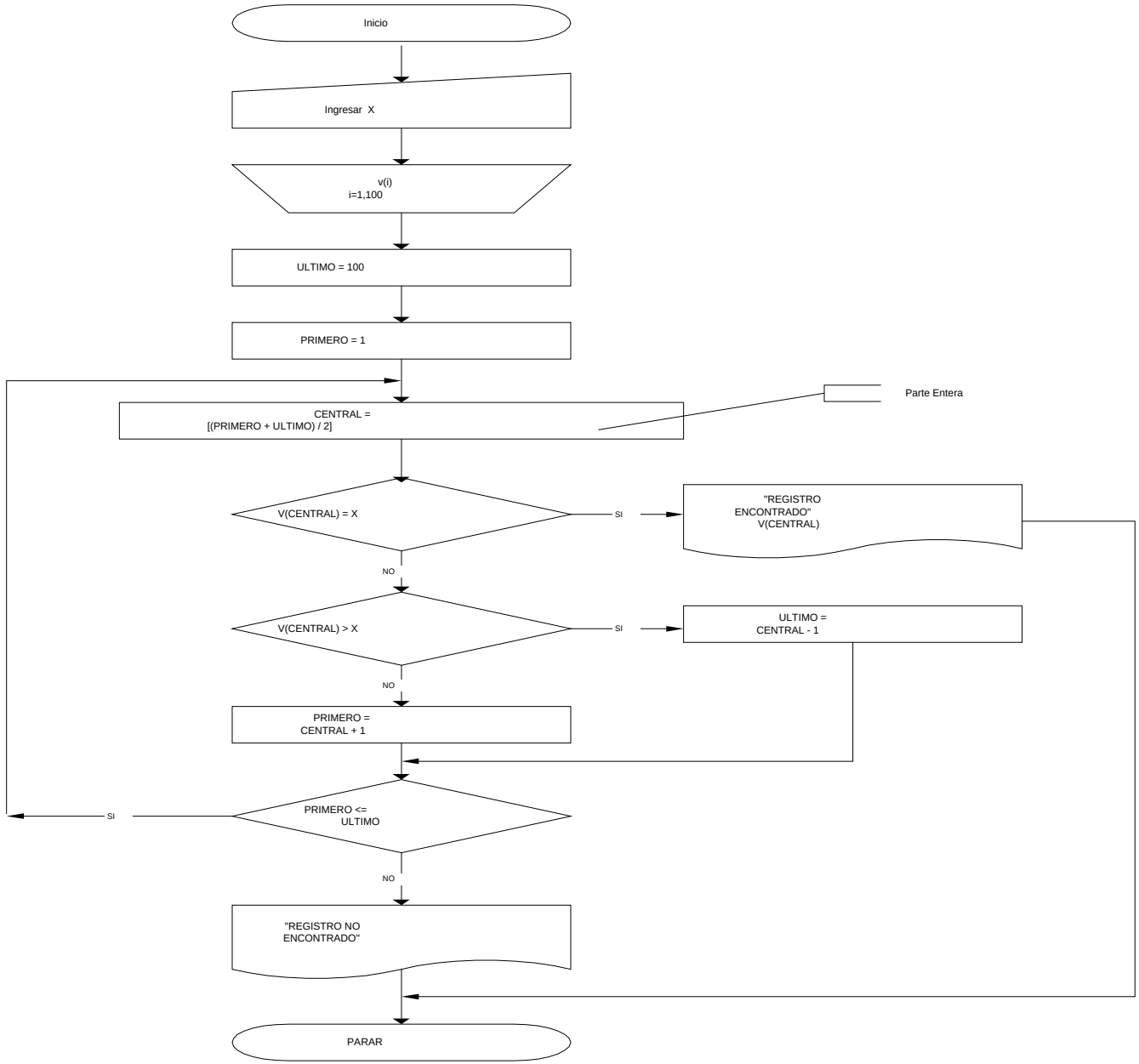
Este mecanismo es muy eficaz para buscar un elemento cualquiera que esté en una lista ordenada, y recibe el nombre de **Búsqueda Binaria o Dicotómica** cuya resolución se base en el algoritmo de divisiones sucesivas en mitades.

A	D	G	H	K	L	N	P	S	W

Buscar el valor G de la lista



Con cada iteración del método el espacio de búsqueda se reduce a la mitad, por lo tanto el número de comparaciones disminuye considerablemente en cada iteración. Esta disminución es mas significativa cuanto mayor sea el número de elementos de la lista.


Inicio
Ingresar X
v(i)
i=1,100
ULTIMO = 100
PRIMERO = 1
CENTRAL =
[(PRIMERO + ULTIMO) / 2]
Parte Entera
V(CENTRAL) = X
SI
"REGISTRO
ENCONTRADO"
V(CENTRAL)
NO
V(CENTRAL) > X
SI
ULTIMO =
CENTRAL - 1
NO
PRIMERO =
CENTRAL + 1
PRIMERO <=
ULTIMO
SI
NO
"REGISTRO NO
ENCONTRADO"
PARAR

- *En Pseudocódigo:*
- **Comenzar**
- **Leer** X
- **Ingresar** V(I) I = 1,100
- ULTIMO = 100
- PRIMERO = 1
- **Hasta** PRIMERO <= ULTIMO
- CENTRAL = [(PRIMERO + ULTIMO) / 2]
- **Si** V(CENTRAL) = X
- **Entonces**
- **Imprimir** “Registro encontrado “ V(CENTRAL)
- **Parar**
- **Fin_si**
- **Si** V(CENTRAL) > X
- **Entonces**
- ULTIMO = CENTRAL – 1
- **Si_no**
- PRIMERO = CENTRAL + 1
- **Fin_si**
- **Fin_Mientras**
- **Imprimir** “Registro no encontrado”
- **Parar**

Código en C: Búsqueda Binaria

```
#include<stdio.h>
int main() {
        int A[20]={21,32,43,54,65,76,87,98,109,110,211,212,313,314,415,416,417,518,519,620};
        Int inf,sup,mit,dato,n=20;
        printf("dame un dato a buscarn: ");
        scanf("%d",&dato);
        inf=0;
        sup=n;
        while (inf<=sup) {
            mit=(inf+sup)/2;
            if (A[mit]==dato) {
                    printf("dato %d encontrado posicion %d \n",dato,mit);
                    break;
                    }
           if (A[mit]>dato) {
              sup=mit;
                mit=(inf+sup)/2;
                }
              if (A[mit]<dato)  {
                inf=mit;
                mit=(inf+sup)/2;
               }
       }
  return 0;
}
```

Búsqueda binaria

- Igual que en el método secuencial la complejidad del método se va a medir por los casos extremos que puedan presentarse en el proceso de búsqueda.

- El caso mas favorable se dará cuando el primer elemento central es el buscado, en cuyo caso se hará una sola comparación. El caso mas desfavorable se dará cuando el elemento buscado no está en las sublistas, en este caso se harán en forma aproximada $\log_2(n)$ comparaciones, ya que en cada ciclo de comparaciones el número de elementos se reduce a la mitad, factor de 2. Por lo tanto, el número medio de comparaciones que se realizarán con este método es de: **$(1 + \log_2(n)) / 2$**

- Si comparamos las fórmulas dadas en ambos métodos (ver apartado 5.1.1. Método secuencial), resulta que para el mismo valor de N el método binario es mas eficiente que el método secuencial; además la diferencia es mas significativa cuanto mas crece N.

Búsqueda mediante transformación de claves (hashing)

- Es un método de búsqueda que aumenta la velocidad de búsqueda, pero que no requiere que los elementos estén ordenados.
- Consiste en asignar a cada elemento un índice mediante una transformación del elemento. Esta correspondencia se realiza mediante una función de conversión, llamada función hash.
- La correspondencia más sencilla es la identidad, esto es, al número 0 se le asigna el índice 0, al elemento 1 el índice 1, y así sucesivamente. (números demasiado grandes, función es inservible).
- La función de hash ideal debería ser biyectiva, esto es, que a cada elemento le corresponda un índice, y que a cada índice le corresponda un elemento, pero no siempre es fácil encontrar esa función, e incluso a veces es inútil, ya que puedes no saber el número de elementos a almacenar.

Por todo lo mencionado, para trabajar con este método de búsqueda debe elegirse previamente:

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

Búsqueda mediante transformación de claves (hashing)

La función de hash depende de cada problema y de cada finalidad, y se pueden utilizar con números o cadenas, pero las más utilizadas son:

- Restas sucesivas
- Aritmética modular
- Mitad del cuadrado
- Truncamiento
- Plegamiento
- Tratamiento de colisiones

Restas sucesivas

Esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas. Por ejemplo, si el número de expediente de un alumno universitario está formado por el año de entrada en la universidad, seguido de un número identificativo de tres cifras, y suponiendo que entran un máximo de 400 alumnos al año, se le asignarían las claves:1998-000 --> 0 = 1998000-1998000

1998-001 --> 1 = 1998001-1998000

1998-002 --> 2 = 1998002-1998000

...

1998-399 --> 399 = 1998399-1998000

1999-000 --> 400 = 1999000-1998000+400

...

yyyy-nnn --> N = yyyynnn-1998000+(400*(yyyy-1998))

ARITMÉTICA MODULAR

- Este método convierte la clave a un entero, se divide por el tamaño del rango del índice y toma el resto como resultado. La función que se utiliza es el MOD(módulo o resto de la división entera).
  - H(x)= x MOD m
- Donde m es el tamaño del arreglo. La mejor elección de los módulos son los números primos.
- Un vector T tiene cien posiciones (0..100). Se tiene que las claves de búsqueda de los elementos de la tabla son enteros positivos. La función de conversión H debe tomar un número arbitrario entero positivo x y convertirlo en un entero en el rango de (0..100)
- H(x)= x MOD m Si clave=234661234 MOD 101 = 56 234661234 MOD 101 = 56

TRUNCAMIENTO

- Ignora parte de la clave y se utiliza la parte restante directamente como índice (considerando campos no numéricos y sus códigos numéricos.
- Ejemplo:
  Se tiene claves de tipo entero de 8 dígitos y para la tabla de transformación tiene mil posiciones, entonces para la dirección(índice) se considera: el primer, segundo y quinto dígito de derecha forman la función de conversión.

clave:72588495 –> h(clave)=728

PLEGAMIENTO

consiste en dividir el número en diferentes partes, y operar con ellas (normalmente con suma o multiplicación). También se produce colisión. Por ejemplo, si dividimos los número de 8 cifras en 3, 3 y 2 cifras y se suman, dará otro número de tres cifras (y si no, se cogen las tres últimas cifras):

13000000 --> 130 =130+000+00
12345678 --> 657 =123+456+78
71140205 --> 118 --> 1118 =711+402+05
13602499 --> 259 =136+024+99
25000009 --> 259 =250+000+09

MITAD DEL CUADRADO

- Este método consiste en calcular el cuadrado de la clave x. La función de conversión se define como:

  H(x)=c. Donde c se obtiene eliminando dígitos a ambos lados de x2.

- Ejemplo:
  Una empresa tiene ochenta empleados y cada uno de ellos tiene un número de identificación de cuatro dígitos y el conjunto de direcciones de memoria varía en el rango de 0 a 100. Se pide calcular las direcciones que se obtendrán al aplicar la función de conversión por la mitad del cuadrado de los números empleados:

  x –> 4205, 7148, 3350, etc.

  x2 –> 17682025 ➔ 82 (dirección de memoria) = H(x)

  x2 -> 51093904 => 93 (dirección de memoria) = H(x)

  x2 -> 11222500 => 22 (dirección de memoria) = H(x)

Tratamiento de colisiones

- Cuando el índice correspondiente a un elemento ya está ocupada, se le asigna el primer índice libre a partir de esa posición. Este método es poco eficaz, porque al nuevo elemento se le asigna un índice que podrá estar ocupado por un elemento posterior a él, y la búsqueda se ralentiza, ya que no se sabe la posición exacta del elemento.

- También se pueden reservar unos cuantos lugares al final del array para alojar a las colisiones. Este método también tiene un problema: ¿Cuánto espacio se debe reservar? Además, sigue la lentitud de búsqueda si el elemento a buscar es una colisión.

- Lo más efectivo es, en vez de crear un array de número, crear un array de punteros, donde cada puntero señala el principio de una lista enlazada. Así, cada elemento que llega a un determinado índice se pone en el último lugar de la lista de ese índice. El tiempo de búsqueda se reduce considerablemente, y no hace falta poner restricciones al tamaño del array, ya que se pueden añadir nodos dinámicamente a la lista

Búsqueda Interpolada

Las condiciones iniciales son las siguientes :

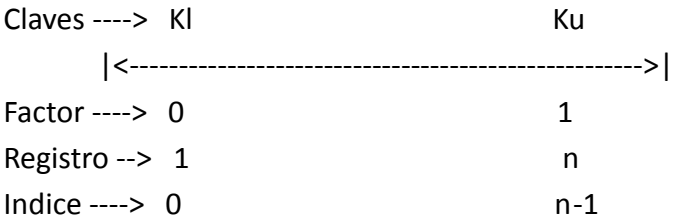
- El archivo donde buscar ya está en memoria, dispuesto en slots contiguos.
- Los registros del archivo los denominaremos R1, R2, R3, ..., Ri, ..., Rn donde 1=< i =< n.
- Existen en memoria n slots, donde en cada uno de ellos se dispone un registro del archivo donde buscar.
- Todos los registros tienen una clave Ki, donde i es 1 =< i =< n.
- Los registros están **ordenados** respecto de la clave mencionada.
- La búsqueda será hecha en los mismos slots donde están los registros.

Búsqueda Interpolada

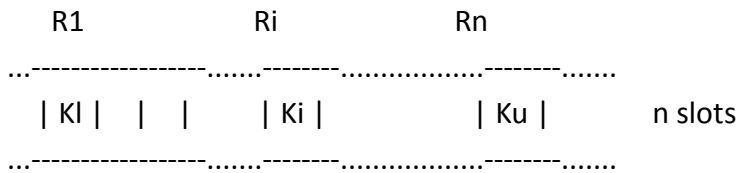
Los criterios de la búsqueda son los siguientes :

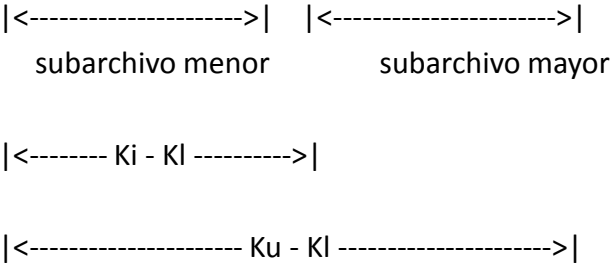
- Ni la búsqueda binaria, ni la búsqueda de has, buscan como lo hace el ser humano, por ejemplo, al buscar en la guía telefónica, una persona cuyo apellido empieza con W.
- Nosotros tenemos idea de una cierta proporcionalidad, respecto del tamaño del archivo, y vamos directamente a la parte alta de la guía telefónica, siguiendo esa idea de proporcionalidad. En el caso de la búsqueda del apellido que comienza con W vamos a la parte final de la guía.
- La idea de la búsqueda interpolada es arriesgar una proporcionalidad basandose en los valores de las siguientes claves :
  - Clave mas baja, que llamamos Kl (l de low)
  - Clave de búsqueda, que llamammos K
  - clave de interpolación, que llamamos Ki
  - Clave mas alta, que llamamos Ku (u de up)
- donde el índice de la clave de interpolación es :
  i = ((K - Kl) / (Ku - Kl)) * n = factor de interpolación * n
  donde :
  - Kl es la clave mas baja (low)
  - Ku es la clave mas alta (up)
  - K es la clave de búsqueda
  - n es la cantidad de claves presentes en el archivo o cantidad de registros
- Observar que si K = Kl ------> factor de interpolación = (Kl - Kl) / (Ku - Kl) = 0 / (Ku - Kl) = 0; donde para un archivo de 70 registros i = 0 * 70 = 0
  Observar que si K = Ku ------> factor de interpolación = (Ku - Kl) / (Ku - Kl) = 1; donde para un archivo de 70 registros i = 1 * 70 = 70

Búsqueda Interpolada



Establecido el ínidce de la clave de interpolación, queda fijado el registro Ri, de clave Ki, y determinados los subarchivos menor y mayor.





- Si K = Ki, la clave fué hallada.
  Si K < Ki, se repite el procedimiento con el subarchivo menor.
  Si K > Ki, se repite el procedimiento con el subarchivo mayor.

Colas de prioridad

- Una cola de prioridades es una estructura de datos en la que los elementos se atienden en el orden indicado por una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen.

  •

  Este tipo especial de colas tienen las mismas operaciones que las colas FIFO, pero con la condición de que los elementos se atienden en orden de prioridad.

  Ejemplos de la vida diaria serían la sala de urgencias de un hospital, ya que los enfermos se van atendiendo en función de la gravedad de su enfermedad.

  Entendiendo la prioridad como un valor numérico y asignando a altas prioridades valores pequeños, las colas de prioridad nos permiten añadir elementos en cualquier orden y recuperarlos de menor a mayor.

Colas de prioridad

Hay 2 formas de implementación:

- Añadir un campo a cada nodo con su prioridad. Resulta conveniente mantener la cola ordenada por orden de prioridad.
- Crear tantas colas como prioridades haya, y almacenar cada elemento en su cola.

Existe 2 tipos de colas de prioridad:

- Colas de prioridades con ordenamiento ascendente: en ellas los elementos se insertan de forma arbitraria, pero a la hora de extraerlos, se extrae el elemento de menor prioridad.

- Colas de prioridades con ordenamiento descendente: son iguales que la colas de prioridad con ordenamiento ascendente, pero al extraer el elemento se extrae el de mayor prioridad.

Bibliografía

Libros

- Fundamentos de programación. Algoritmos, estructuras de datos y objetos; Luis Joyanes Aguilar; 2003; Editorial: MCGRAW-HILL. ISBN: 8448136642.

- PROGRAMACIÓN; Castor F. Herrmann, María E. Valesani.; 2001; Editorial: MOGLIA S.R.L..ISBN: 9874338326.

Link:

https://issuu.com/luiseduardovivar/docs/algbusquedaytransfllaves

http://mtodosdeordenamientoybsqueda.blogspot.com.ar/p/por-enumeracion.html

http://es.slideshare.net/javiervilugron/algoritmo-de-busqueda-truncamiento