Universidad Autónoma de Entre Ríos
Facultad de Ciencia y Tecnología
Sede: Oro Verde

FUNDAMENTOS DE PROGRAMACIÓN

RESUMEN DE CONTENIDOS N°10

Búsqueda y Ordenamiento

RESUMEN DE CONTENIDOS N° 10

Búsqueda y Ordenamiento

INTRODUCCIÓN

Tanto la búsqueda como el ordenamiento, constituyen dos aspectos relevantes dentro del manejo de la información. Estos algoritmos son clásicos y su conocimiento es fundamental para cualquier programador.

La búsqueda está relacionada con procesos de recuperación de información y consiste en localizar un dato determinado dentro de un conjunto.

Dado un arreglo y un valor del mismo tipo que sus componentes, la búsqueda consiste en determinar si ese valor está en el arreglo y qué posición ocupa.

El ordenamiento provee un medio para la organización de la información, facilitando de esta manera la recuperación posterior de los datos. Dado un conjunto de datos, ordenar dicho conjunto, significa obtener un nuevo conjunto donde sus datos se encuentren ordenados ya sea en forma ascendente o descendente.

En ambos casos, existen distintos métodos, cuya eficiencia está directamente relacionada con él número de comparaciones y el movimiento de los datos llevados a cabo para completar la operación.

BÚSQUEDA

Dado un conjunto de datos se trata de encontrar un dato particular dentro del mismo. El dato que se quiere buscar, puede o no estar entre los elementos del conjunto. Esta operación es muy frecuente dentro de los procedimientos algorítmicos, por ejemplo, encontrar un artículo determinado entre todos los que figuran en la lista de precios de una empresa.

El problema de búsqueda puede plantearse de la siguiente forma:

Dado un conjunto de N elementos distintos y un dato K, determinar:

- si K pertenece al conjunto y su ubicación dentro del mismo

- si K no pertenece al conjunto.

Para resolver este problema se estudiarán dos métodos:

1) Búsqueda Secuencial o Lineal

2) Búsqueda Binaria o Dicotómica

1) Búsqueda Secuencial o Lineal

Este método consiste en recorrer, de izquierda a derecha, todo el conjunto, comparando K con cada uno de los elementos del mismo. El proceso finaliza al encontrar el dato buscado o al llegar al final del conjunto, en cuyo caso se puede concluir que K no figura en el conjunto dado. En los casos en que pueda aparecer el valor repetido, el algoritmo indica el de índice menor (en otras palabras, el primer elemento que coincida con K). Este método es válido tanto para vectores desordenados como ordenados.

Al encontrar el elemento debemos suspender la búsqueda y salir del ciclo. Es necesario establecer en el algoritmo cómo se ha salido del ciclo de búsqueda. Es decir, si fue por haber encontrado el dato buscado o por haber llegado al final del arreglo sin encontrar el dato. Esto

puede lograrse dando diferentes valores a la variable de control del ciclo.

Este algoritmo requiere una sola comparación si el elemento buscado es el primer elemento del conjunto, o N comparaciones, en el peor de los casos, si es el último o no pertenece al conjunto. En promedio, se requieren (N+1)/2 comparaciones.

En C++, un algoritmo de búsqueda secuencial puede plantearse de la siguiente manera:

```
#include <iostream>
using namespace std;
//Búsqueda Secuencial

int main() {
   int A[10], i, band=0, dat, pos, n=5;

   for (i=0; i<n; i++)
   {
      cout<<"Dato: "; cin>>A[i];
   }
   cout<<"****************************************"<<endl;
   cout<<"Ingrese el dato a buscar: ";
   cin>>dat;
   i=0;
   while (i < n)
   {
      if (A[i] == dat)
      {     band=1;
            pos=i+1;
            i=n;
      }
      i++;
   }
   cout<<"****************************************"<<endl;

   if (band == 0)
      cout<<"El dato no fue hallado";
   else cout<<"El dato fue hallado en la posición "<<pos;

   return 0;
}
```

Como podrá observarse, en el algoritmo anterior puede reducirse el uso de variables auxiliares, pero se pretenden mostrar de manera clara los diferentes puntos a considerar

Búsqueda Secuencial o Lineal en una Matriz

El método de búsqueda visto, puede aplicarse también a un conjunto de datos dispuestos en una matriz. En este caso, se realiza mediante el anidamiento de dos estructuras iterativas, cuya finalización vendrá dada por la aparición del valor buscado o la finalización de la matriz.

Usualmente se comienza recorriendo la matriz por filas, aunque cambiando de posición los índices con sus correspondientes límites, se puede hacer igualmente por columnas.

2) Búsqueda Binaria o Dicotómica

Este método sólo puede aplicarse a conjuntos cuyos elementos están *ordenados*, ya sea en forma ascendente o descendente. También se lo conoce con el nombre de búsqueda por bisección o dicotómica.

Para la explicación del método, se supondrá que se quiere buscar un valor en un vector ordenado en forma ascendente (de menor a mayor).

El método consiste en partir sucesivamente por mitades el vector y preguntar si el valor buscado es el elemento ubicado en la mitad de la tabla. Si la respuesta es positiva se finaliza la búsqueda. Si no, el valor buscado debe estar en una de las dos mitades del vector con lo cual se descartan todos los elementos de la otra mitad. Con este segmento del vector donde posiblemente está el elemento buscado, se repite el proceso descripto, partiéndolo por la mitad y preguntando si el valor buscado es igual al contenido del elemento de la mitad. Este proceso se repite hasta que se encuentre el valor buscado o hasta que se llegue al límite de consultas en cuyo caso, el valor buscado no está en el vector.

En los casos en que existan repeticiones en el vector del valor buscado, este algoritmo obtendrá uno de ellos aleatoriamente según los lugares que ocupen, los cuales necesariamente son consecutivos.

Descripción del Método

El primer paso consiste en encontrar aproximadamente, la posición media del vector y examinar el valor que contiene. Si A es el vector, entonces se compara el dato buscado K con A[medio].

Al hacer dicha comparación, puede suceder:

a) K = A[medio]

Finaliza la búsqueda y el elemento buscado se encontró, en la posición “medio”.

b) K < A[medio]

En este caso, se considera la primera mitad de la tabla como la próxima tabla de búsqueda. En esta nueva tabla, se calcula nuevamente el elemento situado en la posición central y se repite todo el proceso presentándose nuevamente una de las tres opciones posibles (a, b ó c).

c) K > A[medio]

Se considera la segunda mitad de la tabla y se procede como en el caso b).

El procedimiento se repetirá en forma iterativa hasta hallar el elemento buscado en alguna de las sucesivas iteraciones, o hasta agotar la tabla (cuando el intervalo de búsqueda queda vacío), en cuyo caso dicho elemento no pertenece al conjunto dado.

La posición media del vector se determina en cada paso como la parte entera de la

siguiente expresión:

(valor inferior del índice + valor superior del índice) / 2

En C++:

```
int medio;
medio=(0+N-1)/2;
```

En la expresión precedente, los valores inferiores y superiores del índice se calculan en cada nueva iteración.

Ejemplo Dado el siguiente conjunto de 8 elementos ordenados en forma ascendente

5	12	15	17	22	27	33	38
0	1	2	3	4	5	6	7

El elemento a buscar es K = 33

- Determinar primero la posición media del vector:

```
medio= (0+7)/2;     // C++ hace división entera si opera con enteros.
```

El dato ubicado en dicha posición es `A[3]=17`

- Como K=33 es mayor que 17, se eliminan los elementos `A[0], A[1]` y `A[2]` y se continúa la búsqueda con los elementos de la segunda mitad: `A[4], A[5], A[6]` y `A[7].`

- Se determina el elemento central de esta nueva tabla, donde el valor del índice inicial es 4 y el final 7. Por lo tanto:

```
medio = (4+7)/2) = 5
```

- Y el elemento central es ahora `A[5]=27.`

Como el valor K es mayor que `A[5]` se deben analizar los elementos que están a su derecha: `A[6]` y `A[7].`

- Para hallar el elemento central de este nuevo conjunto, se debe considerar el valor del índice Inicial 6 y el final 7. Luego, la posición media está dada por:

```
medio = (6+7)/2 = 6 ==> A[medio] = A[6] =33
```

- Como `A[6])` es igual a K, la búsqueda finaliza determinando que el elemento está en la posición 6.

Observar en el ejemplo que, en cada paso se analiza un vector, para el cual se definen diferentes valores extremos del índice. Por lo tanto, para plantear el algoritmo se necesita:

- una variable, para guardar el valor inicial del índice del vector que se está analizando;
- una segunda variable para el valor superior del índice del vector que se está analizando;
- una tercer variable, para guardar el valor de la posición **medio.**

En el primer paso, el valor inicial es 0 y el superior es N-1. Luego dependerán del vector que se considere para la búsqueda.

Observación: Deberá recordarse que en C++, los arreglos tienen dimensión N y el índice varía entre 0 y N-1. En otros lenguajes, el índice del arreglo varía entre 1 y N.

Pero, ¿hasta cuándo se debe consultar la tabla?.

Existe un número máximo de particiones, agotado el cual se puede suponer que el valor buscado no pertenece a la misma. Este máximo está dado por la menor potencia de 2 que excede al número de elementos del vector que se consulta. Así, si el vector tiene 500 elementos, el número máximo de particiones es 9, porque 9 es la mínima potencia de 2 que excede a 500.

En C++, un algoritmo de búsqueda binaria puede plantearse de la siguiente manera:

```
#include <iostream>
using namespace std;
//Búsqueda Binaria

int main() {
   int li, ls, n, medio, dat, A[10], i;

   cout<<"Ingrese la cantidad de datos a leer: "; cin>>n;
   for (i=0; i<n; i++)
   {
       cout<<"Dato: "; cin>>A[i];
   }
   cout<<"****************************************"<<endl;
   cout<<"Ingrese el dato a buscar: ";
   cin>>dat;
   li=0; ls=n; medio= (li+ls)/2;
   while ((li <= ls) && (dat != A[medio]))
   {
       if (dat < A[medio])
             ls=medio-1;
        else li=medio+1;
      medio= (li+ls)/2;
   }
   cout<<"****************************************"<<endl;
   if (li > ls)
      cout<<"Elemento no encontrado";
    else
      cout<<"Elemento encontrado en posición: "<<medio+1;
   return 0;
}
```

ORDENAMIENTO

En los métodos de ordenamiento, además de considerar el tiempo de ejecución del mismo, se debe tener en cuenta el uso eficiente de la memoria.

Entre los métodos de ordenamiento conocidos, es posible hacer una clasificación sobre la base de su eficiencia. Ello indica que los métodos avanzados son mucho más eficientes. Pero antes de conocer estos algoritmos más rápidos, en esta materia se estudiará un método directo.

Los métodos directos, poseen las siguientes características:

- Son más sencillos y obvios.
- Permiten analizar los principios de clasificación.
- Los algoritmos son cortos y fáciles de entender.
- Los métodos avanzados, si bien requieren menor número de operaciones, se basan en algoritmos complejos y, para arreglos pequeños, los métodos directos son más rápidos.

Los tipos de ordenamientos que pueden realizarse son:

- ***Ordenamiento ascendente o creciente***: Consiste en situar los valores mayores al final (o la derecha) de un arreglo y los menores al principio (o a la izquierda). Los valores repetidos quedarán en posiciones consecutivas.

- ***Ordenamiento descendente o decreciente***: Es el ordenamiento inverso al anterior.

MÉTODO DIRECTO DE ORDENAMIENTO

Selección Directa o Mínimos Sucesivos

El problema planteado para la explicación del método es: Dado un vector A de N elementos, ordenar sus elementos en forma ascendente.

Este método consiste en colocar en la primera posición del vector el menor de los elementos del mismo; luego en la segunda posición, el menor de los N-1 elementos restantes; proseguir con los N-2 elementos restantes seleccionando el menor para la tercera posición y, así sucesivamente hasta que solamente quede el mayor de todos los elementos en la última posición.

Para ello, en la primer recorrida (o pasada) del vector, se debe comparar el contenido del primer elemento con el valor del segundo elemento, y si este último resulta menor, intercambiar valores de tal manera que en la primera posición siempre quede el menor. Luego repetir este proceso entre el primero y el tercero, luego entre el primero y el cuarto y así sucesivamente, hasta comparar el primer elemento con el último. De esa forma, se tendrá la seguridad que en la primera posición se deja el menor de todos los elementos del arreglo.

Se consideran luego los N-1 elementos restantes, desde el segundo elemento hasta el último. Si se realiza el proceso de comparación dado para la primer pasada pero ahora entre el segundo elemento y todos los restantes, se obtendrá en la segunda posición del vector, el menor entre todos los elementos analizados.

Para ordenar todo el vector, se repite este proceso sucesivamente hasta comparar finalmente el penúltimo elemento contra el último, después de lo cual termina el proceso.

Ejemplo: Dado el siguiente vector A de dimensión N=6

12	10	17	9	14	8
0	1	2	3	4	5

La primer pasada al comparar `A[0]` con cada uno de los elementos del vector, resultan los siguientes intercambios:

con `A[1]` 10 12 17 9 14 8

con `A[2]` 10 12 17 9 14 8 (sin cambios)

con `A[3]` 9 12 17 10 14 8

con `A[4]` 9 12 17 10 14 8 (sin cambios)

con `A[5]` 8 12 17 10 14 9

Se obtiene un vector que contiene en la primera posición el menor de todos sus valores.

Ahora, partiendo de este último vector, se debe realizar el mismo proceso para dejar en `A[2]` el menor de todos los elementos. Al comparar `A[2]` con cada uno de los elementos restantes se obtendrá el siguiente vector

8	9	17	12	14	10

En las sucesivas pasadas quedan los siguientes vectores

8	9	10	17	14	12
8	9	10	12	17	14
8	9	10	12	14	17

En C++, un algoritmo de ordenamiento por mínimos sucesivos puede plantearse de la siguiente manera:

```
#include <iostream>
using namespace std;
//Ordenamiento por mínimos sucesivos

int main() {
   int n, A[50], i, j, aux;

   cout<<"Ingrese la cantidad de datos a leer: "; cin>>n;

   for (i=0; i<n; i++)
   {
      cout<<"Dato: "; cin>>A[i];
   }

   for (i=0; i<(n-1); i++)
      for (j=i+1; j<n; j++)
      {
            if (A[i] > A[j])
                  {
                        aux= A[i];
                        A[i]= A[j];
                        A[j]= aux;
                  }
      }

   cout<<endl;
   cout<<"ARREGLO ORDENADO"<<endl;
   for (i=0; i<n; i++)
      cout<<vec[i]<<endl;

    return 0;
}
```

Observación: Deberá recordarse que en C++, los arreglos tienen dimensión N y el índice varía entre 0 y N-1.

ATENCIÓN!: Si al ordenar un arreglo existen otros arreglos relacionados con éste, al momento de intercambiar valores, deberán intercambiarse todos los arreglos relacionados para que la información guardada siga siendo consistente.