Algoritmos de búsqueda con ejemplos en C++

Miguel Iván Bobadilla

Índice

Introducción

Un algoritmo es una secuencia de pasos para resolver un problema previamente planteado, la cual puede ser computacional o no computacional. En este caso nos concentraremos en los algoritmos computacionales que nos han facilitado las tareas en su implementación electrónica. Entre esas tareas tenemos un problema muy común en cualquier ambiente de trabajo, la cual es la de búsqueda de información de manera rápida. Es por ello que los sistemas computacionales -especialmente aquellos que manejan datos- deben poseer algoritmos para búsqueda. Un ejemplo sencillo es cuando se necesita acceder a los datos de un producto o cliente, esto llevaría mucho tiempo, la cual en una empresa el tiempo se valora como dinero, por eso necesitan una manera de acceder a los datos de forma rápida. Con un algoritmo de búsqueda este problema se resolvería ya que agrega optimización y así el usuario no tenga que buscar de forma manual los datos y perderse mucho tiempo valioso.

Por la razón anteriormente mencionado de la optimización del tiempo para acceder a datos es que es importante saber los diferentes métodos o algoritmos de búsqueda para su implementación y así facilitar que los sistemas que se construyan resuelvan estos problemas, que es común en cualquier área de trabajo.

Algoritmos de búsqueda

Un algoritmo de búsqueda como lo dice su nombre es una secuencia de pasos para encontrar dentro de una lista de datos un dato determinado. Los algoritmos de búsqueda varían según los pasos o métodos que emplea para encontrar datos, pero todos cumplen con el mismo fin. Los algoritmos de búsqueda empleados frecuentemente son la búsqueda binaria, búsqueda lineal o secuencial y la búsqueda por transformación de clave o Hash.

Cada algoritmo de búsqueda presentaremos un ejemplo en el lenguaje de programación C++ para tener una imagen clara de cómo el algoritmo resuelve el problema.

Búsqueda lineal o secuencial

Es el algoritmo de búsqueda más simple, pero el menos eficiente, no requiere que los datos o elementos estén ordenados. Consiste en recorrer los registros o arreglos de manera secuencial, es decir, recorriendo elemento por elemento comparando los datos con la clave de búsqueda hasta que encuentre el dato solicitado o determine que dicho dato no se encuentra. El inconveniente con este algoritmo es que si el arreglo posee unas dimensiones muy grandes si el elemento a buscar está muy lejos tardaría más tiempo en encontrarlo.

Supongamos que, tenemos un arreglo de tamaño de cinco posiciones, la cual contiene los siguientes elementos [2,6,7,4,3] y deseamos buscar el elemento 4 en dicho arreglo. Lo que hace el algoritmo es comenzar desde la primera posición (que puede iniciar con 1 o 0) y comparar la clave de búsqueda -en este caso el 4- con la primera posición, si no lo encuentra entonces continua con la segunda, en caso de que aún no lo encuentre repite el proceso de comparar con los siguientes elementos. Dependiente el tipo de ciclo que se utilice para recorrer el arreglo podemos hacer que se detenga cuando lo encuentre o que continúe comparando los demás elementos por si queremos saber si se repite, pero con la advertencia de que la ejecución tardaría más tiempo en terminal hasta llegar al final.

El algoritmo compararía los elementos del arreglo [2,6,7,4,3] de la siguiente manera:

- [2,6,7,4,3]
- ¿2==4?
- No
- [2,6,7,4,3]
- ¿6==4?
- No

- [2,6,7,4,3]
- ¿7==4?
- No
- [2,6,7,4,3]
- ¿4==4?
- Si
- ¡Elemento encontrado!

Ejemplo en C++

```
//Librerias
#include "stdafx.h"
#include <iostream>
using namespace std;

int main()
{
	//Declarando variables.
	int clave,encontrado, n, i, j;
	//Pedir tamaño del arreglo.
	cout << "Ingrese dimensión del arreglo: ";
	cin >> n;
	Cout<<endl;
	//Crear la variable para el arreglo dinámico.
	int* arreglo = new int[n];
	//LLenar el arreglo.
	for (i = 0;i < n;i++)
	{
		cout << "Ingrese dato [" << i << "]: ";
		cin >> arreglo[i];
	}
	//Pedir dato o clave a buscar.
	cout << "Ingrese dato que desea buscar: " << endl;
	cin >> clave;
	//Declarar variable de éxito de búsqueda.
	encontrado = 0;
	//Buscar clave o dato en el arreglo.
	for (j = 0;j < n;j++)
	{
		//Si el elemento de la posición actual del arreglo es
		similar a la clave de búsqueda, despliega mensaje de
		encontrado.
		if (arreglo[j] == clave)
		{
```

```
                cout << "Se encontro el " << clave << " en la
                posicion [" << j << "]" << endl;
                encontrado = 1;
            }
        }
        //Libera arreglo dinámico en memoria.
        delete[] arreglo;
        //Si no se encontró la clave despliega el mensaje de no
        encontrado.
        if (encontrado != 1)
            cout << "No se encontro el dato" << endl;
        system("pause");
        return 0;
}
```

Búsqueda binaria

Es un algoritmo de búsqueda más eficiente que el anterior, pero posee el inconveniente que el arreglo debe estar ordenado. Su metodología consiste en colocarse en medio o parte central del arreglo, si no encuentra el elemento en dicha posición entonces se pregunta si el valor de dicha posición es menor o mayor a la clave de búsqueda, en caso de que sea menor entonces se coloca en medio del primer bloque de elementos del arreglo e ignora los elementos del segundo bloque, en caso de que sea mayor se coloca en medio del segundo bloque de elementos del arreglo e ignora los elementos del bloque anterior. Estos pasos se repiten hasta que se encuentre el elemento buscado.

Lo que hace que este algoritmo sea mucho más eficiente que el algoritmo de búsqueda secuencial se debe a que en vez de pasar uno por uno los elementos de todo el arreglo, lo que hace es ignorar los bloques de elementos donde indudablemente se sabe –debido a que está ordenado- que no puede encontrarse allí. Por ejemplo, imaginemos que tenemos un arreglo que contiene 11 elementos distribuidos de manera ordenada [2,6,9,12,15,18,24,29,32,45,50] y queremos buscar el elemento 12 en dicho arreglo. El algoritmo divide el arreglo en dos partes o bloques y se coloca en la posición del medio, en este caso se colocará en el elemento 18 ya que es la posición media del arreglo. Luego compara dicho elemento con la clave de búsqueda o dato que deseamos buscar, lo cual resulta que 18 no es similar a 12, por lo cual procede a verificar si dicho elemento es menor o mayor a la clave de búsqueda, es decir, si 18 es menor o mayor a 12, en este caso 18 es mayor a 12, por la cual el algoritmo procede a ignorar todos los elementos mayores e iguales que 18, es decir, ignora todos los elementos del segundo bloque y repite el proceso centrándose en el primer bloque y dividiendo dicho bloque en dos partes. En caso de que

no encuentre el elemento repite el proceso hasta que indique si el elemento se encuentre o no en el arreglo.

El algoritmo compararía los elementos del arreglo [2,6,9,12,15,18,24,29,32,45,50] de la siguiente manera:

- [2,6,9,12,15,18,24,29,32,45,50]
- ¿18==12?
- No
- 18 > 12
- Se ignora segundo bloque (18,24,29,32,45,50)
- [2,6,9,12,15]
- ¿9==12?
- No
- 9 < 12
- Se ignora primer bloque (2,6,9)
- [12,15]
- ¿12==12?
- Si
- ¡Elemento encontrado!

Ejemplo en C++

```
#include "stdafx.h"
#include <iostream>
using namespace std;


int main()
{
	//Declarar variables.
	int medio, inicio, fin, valormedio, clave, n, encontrado;
	int temp, i, j, m, p;
	inicio = 0;
	//Pedir tamaño de arreglo.
	cout << "Ingrese dimension del arreglo: ";
	cin >> n;
	cout<<endl;
	fin = n - 1;
	//Crear la variable para el arreglo dinámico.
	int* arreglo = new int[n];
	//Llenar arreglo.
	for (i = 0;i < n;i++)
```

```
{
      cout << "Ingrese dato [" << i << "]: ";
      cin >> arreglo[i];
}
//Formatear siguientes variables con cero.
temp = 0;
j= 0;
m = 0;
p = 0;
//Ordenar arreglo por burbuja.
for (j = 1; j < n; j++)
{
      for (m = 0; m < n-1; m++)
      {
            if(arreglo[m]>arreglo[m + 1])
            {
                  temp = arreglo[m + 1];
                  arreglo[m + 1] = arreglo[m];
                  arreglo[m] = temp;
            }
      }
}
//Mostrar arreglo ordenado.
Cout<<endl;
cout << "El arreglo Ordenado es :" << endl;
for (p = 0; p <= n - 1; p++)
{
      cout << "Posicion [" << p << "] = " << arreglo[p] <<
      endl;
}
//Pedir dato o clave a buscar.
cout << "Ingrese dato que desea buscar: " << endl;
cin >> clave;
//Buscar clave o dato deseado.
encontrado = 0;
while (inicio <= fin)
{
      //Medio será igual a la mitad de la dimensión del
      arreglo.
      medio = (fin + inicio) / 2;
      //Si variable medio es similar a la clave de búsqueda
      despliega mensaje de encontrado.
      if (arreglo[medio] == clave)
      {
            cout << "Se encontro el " << clave << " en la
            posicion [" << medio << "]" << endl;
```

```
                encontrado = 1;
                fin = -1;
            }
            //De lo contrario verifica si la posición del valor
            medio actual es menor que la clave de búsqueda.
            Else
                if (clave < arreglo[medio])
                    fin = medio - 1;
                else
                    inicio = medio + 1;
        }
        //Si no se encontró la clave despliega el mensaje de no
        encontrado.
        if (encontrado != 1)
            cout << "No se encontro el dato" << endl;
        //Libera arreglo dinámico en memoria.
        delete[] arreglo;
        system("pause");
    return 0;
}
```

Búsqueda por transformación de clave (Hash)

La búsqueda por transformación de claves, también llamada búsqueda Hash es más rápida que las anteriores, ya que su velocidad no depende del tamaño del arreglo y no requiere que los elementos estén ordenados. Tampoco tiene que recorrer el arreglo completo como lo hace la búsqueda secuencial ni comparar demasiados elementos como ocurre con la búsqueda binaria. Sin embargo, la búsqueda hash tiene el inconveniente de que si el tamaño de los elementos crece hay que ampliar el espacio de la tabla hash, lo que la hace una operación muy costosa, además si se reserva espacio para todos los posibles elementos, se consume más memoria de lo necesaria; se suele resolver reservando espacio únicamente para punteros a los elementos.

El algoritmo fundamentalmente consiste en convertir o transformar una clave en una dirección (índice) dentro del arreglo. La función de hash depende de cada problema y de cada finalidad, y se pueden utilizar con números o cadenas.

Ejemplo en C++

```
#include "stdafx.h"
#include <iostream>
using namespace std;
typedef struct no no_hash;
```

```
struct no
{
	int data; //Valor del elemento.
	int state; //Estados del elemento: 0 para VACIO, 1 para
REMOVIDO e 2 para OCUPADO.
};
// Calcula la función de distribución
int funcion(int k, int m, int i)
{
	return ((k + i) % m);
}
// Crea la tabla Hash.
no_hash *Crea_Hash(int m)
{
	no_hash *temp;
	int i;
	if ((temp = (no_hash*)malloc(m*sizeof(no_hash))) != NULL)
	{
		//Asigna al arreglo temporal estados vacios.
		for (i = 0; i < m; i++)
			temp[i].state = 0;
		return temp;
	}
	else
		exit(0);
}
// Inserta un elemento k en la tabla T de dimension m.
void Inserta_Hash(no_hash *T, int m, int k)
{
	int j, i = 0;
	do
	{
		j = funcion(k, m, i);
		//Si posicion j de la tabla está vacio || removido...
		if (T[j].state == 0 || T[j].state == 1)
		{
			//Asignale el valor k y coloca su estado como
			ocupado.
			T[j].data = k;
			T[j].state = 2;
			return;
		}
		else
			i++;
	}
	while (i < m);
```

```
	cout << "Tabla llena!"<<endl;
}
//Busca elemento k en la tabla de Hash
int Busca_Hash(no_hash *T, int m, int k, int i)
{
	int j;
	if (i < m)
	{
		j = funcion(k, m, i);
		//Si el estado de la posicion j en la tabla está vacio.
		if (T[j].state == 0)
			return -1;
		//De lo contrario si la posición j en la tabla está
		removido.
		else
			if (T[j].state == 1)
				return Busca_Hash(T, m, k, i + 1);
			//De lo contrario si clave de busqueda y elemento
			de tabla coinciden
			//despliega mensaje de encontrado.
			else
				if (T[j].data == k)
				{
					cout<< "Numero "<< k << " encontrado en
					la posicion ["<<j<<"]"<<endl;
					return j;
				}
				else
					return Busca_Hash(T, m, k, i + 1);
	}
	return -1;
}
//Programa principal.
void main()
{
	//Declarar variables.
	int m, i, k, p,c;
	no_hash *T;
	//Pedir tamaño del arreglo.
	cout << "Ingrese dimension del arreglo: ";
	cin >> m;
	cout << endl;
	//Crear arreglo dinamico.
	int* arreglo = new int[m];
	//Crea la tabla de Hash.
	T = Crea_Hash(m);
```

```
	//Ingreso de elementos a la tabla.
	for (c = 0;c < m;c++)
	{
		cout << "Ingrese dato [" << c << "]: ";
		cin >> arreglo[c];
		k = arreglo[c];
		Inserta_Hash(T, m, k);
	}
	cout << endl;
	//Mostrar tabla de Hash.
	cout << "Mostrar tabla de Hash" << endl;
	cout << endl;
	for (p = 0;p < m;p++)
	{
		cout << "Posicion ["<<p<<"]: "<<T[p].data << endl;
	}
	cout << endl;
	//Pedir elemento de busqueda (k).
	cout << "Ingrese el numero a ser buscado: ";
	cin >> k;
	i = Busca_Hash(T, m, k, 0);
	//Si no lo encuentra despliega mensaje.
	if (i == -1)
		cout << "Numero no encontrado!"<<endl;
	//Liberar memoria de arreglo dinamico.
	delete[] arreglo;
	system("pause");
}
```

Bibliografía

[En línea] https://www.fing.edu.uy/tecnoinf/mvd/cursos/prinprog/material/teo/prinprog-teorico11.pdf.

[En línea] http://novella.mhhe.com/sites/dl/free/844814077x/619434/A06.pdf.

Contreras, Jorge . Análisis de Algoritmos. [En línea] https://jorgecontrerasp.wordpress.com/unidad-i/hashing/.

Estructura de datos ITP. [En línea] https://estructuradedatositp.wikispaces.com/6.3.+M%C3%A9todo+de+b%C3%BAsqueda+POR+FUNCIONES+DE+HASH.

Euán Ávila , Jorge Iván y Cordero Borboa, Luis Gonzaga. *Métodos de búsqueda.*

Laza Fidalgo, Rosalía. *Algoritmos de Búsqueda y Ordenación.*