



Tecnológico Nacional de México
Instituto Tecnológico Superior De La Región Sierra
Ingeniería Informática



Materia

Estructura de datos

Actividad

Investigación Unidad 5 y 6

Presenta

José Alejandro Cepulveda Fonseca

Docente

Jesús Manuel May León

Fecha y lugar

Teapa Tabasco a 23 de noviembre de 2024

Índice

| | |
|---|----|
| Resumen | 3 |
| Introducción | 4 |
| Algoritmos de Ordenamiento..... | 5 |
| Método De Ordenamiento Burbuja | 5 |
| Método Quick sort | 6 |
| Método De Ordenamiento Shell Sort | 7 |
| Método De Ordenamiento Radix | 9 |
| Algoritmos de búsqueda | 11 |
| Búsqueda secuencial | 11 |
| Búsqueda Binaria | 12 |
| Búsqueda por función hash | 14 |
| Conclusión | 16 |
| Bibliografía | 17 |

Resumen

Algoritmos de Ordenamiento:

- **Burbuja:** Compara pares adyacentes y los intercambia si están en el orden incorrecto. Simple pero ineficiente para grandes conjuntos de datos.
- **QuickSort:** Divide la lista en sublistas y ordena recursivamente cada sublista. Generalmente muy eficiente.
- **ShellSort:** Mejora del ordenamiento por inserción, utilizando incrementos decrecientes para reducir el número de comparaciones.
- **RadixSort:** Ordena elementos basándose en sus dígitos, evitando comparaciones directas. Eficiente para números enteros.

Algoritmos de Búsqueda:

- **Secuencial:** Compara cada elemento de la lista hasta encontrar el objetivo. Simple pero ineficiente para listas grandes.
- **Binaria:** Divide la lista a la mitad repetidamente hasta encontrar el objetivo. Muy eficiente para listas ordenadas.
- **Hash:** Utiliza una función hash para asignar elementos a ubicaciones específicas en una tabla. Extremadamente rápido para búsquedas, pero requiere una buena función hash.

Introducción

Los algoritmos de ordenamiento y búsqueda son herramientas fundamentales en la programación que permiten manipular y organizar datos de manera eficiente. Los algoritmos de ordenamiento reordenan elementos de una lista según un criterio específico, mientras que los algoritmos de búsqueda localizan elementos particulares dentro de una estructura de datos

Algoritmos de Ordenamiento

Método De Ordenamiento Burbuja

El ordenamiento de burbuja (Bubble Sort en inglés) es un sencillo algoritmo de ordenamiento. Este método consiste en ir comparando cada par de elementos del array e ir moviendo el mayor elemento hasta la última posición, comenzando desde la posición cero. Una vez acomodado el mayor elemento, prosigue a encontrar y acomodar el segundo más grande comparando de nuevo los elementos desde el inicio de la lista, y así sigue hasta ordenar todos los elementos del arreglo. Al ser necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, hace que el ordenamiento por burbuja sea uno de los algoritmos más ineficientes que existen.

Estos serían los pasos a seguir por este algoritmo para ordenar una lista $a_1, a_2, a_3, \dots, a_n$

- 1) Comparar a_1 con a_2 e intercambiarlos si $a_1 > a_2$
- 2) Seguir hasta que se haya comparado a_{n-1} con a_n
- 3) Repetir el proceso anterior $n-1$ veces

Ejemplo en código C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Burbuja
{
    class Burbuja
    {
        private int[] vector;

        public void Cargar()
        {
            Console.WriteLine("Metodo de Burbuja");
            Console.Write("Cuantos longitud del vector: ");
            string linea;
            linea = Console.ReadLine();
            int cant;
            cant = int.Parse(linea);
            vector = new int[cant];
            for (int f = 0; f < vector.Length; f++)
            {
                Console.Write("Ingrese elemento " + (f + 1) + ": ");
                linea = Console.ReadLine();
                vector[f] = int.Parse(linea);
            }
        }
    }
}
```

```
public void MetodoBurbuja()
{
    int t;
    for (int a = 1; a < vector.Length; a++)
        for (int b = vector.Length - 1; b >= a; b--)
        {
            if (vector[b - 1] > vector[b])
            {
                t = vector[b - 1];
                vector[b - 1] = vector[b];
                vector[b] = t;
            }
        }
}

public void Imprimir()
{
    Console.WriteLine("Vector ordenados en forma ascendente");
    for (int f = 0; f < vector.Length; f++)
    {
        Console.Write(vector[f] + " ");
    }
    Console.ReadKey();
}

static void Main(string[] args)
{
    Burbuja pv = new Burbuja();
    pv.Cargar();
    pv.MetodoBurbuja();
    pv.Imprimir();
}
```

```
Metodo de Burbuja
Cuantos longitud del vector: 8
Ingrese elemento 1: 9
Ingrese elemento 2: 7
Ingrese elemento 3: 0
Ingrese elemento 4: 3
Ingrese elemento 5: 1
Ingrese elemento 6: 2
Ingrese elemento 7: 65
Ingrese elemento 8: 100
Vector ordenados en forma ascendente
0 1 2 3 7 9 65 100
```

Resultado del codigo

Método Quick sort

El método de ordenamiento Quick Sort es actualmente el más eficiente y veloz de los métodos de ordenación interna. Es también conocido con el nombre del método rápido y de ordenamiento por partición, en el mundo de habla hispana.

Este método es una mejora sustancial del método de intercambio directo y recibe el nombre de Quick Sort por la velocidad con que ordena los elementos del arreglo. Su autor C.A. Hoare lo bautizó así.

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, que son menores o iguales al pivote y otra por los elementos a su derecha que son mayores o iguales al pivote.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Ejemplo en código c#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace quicksort
{
    class Class
    {
        static void Main()
        {
            int n;
            Console.WriteLine("Metodo de Quick Sort");
            Console.Write("Cuantos longitud del vector: ");
            n = Int32.Parse(Console.ReadLine());
            llenar b = new llenar(n);
        }
    }

    class llenar
    {
        int h;
        int[] vector;
        public llenar(int n)
        {
            h = n;
            vector = new int[h];
            for (int i = 0; i < h; i++)
            {
                Console.Write("ingrese valor {0}: ", i + 1);
                vector[i] = Int32.Parse(Console.ReadLine());
            }
            quicksort(vector, 0, h - 1);
            mostrar();
        }
    }

    private void quicksort(int[] vector, int primero, int ultimo)
    {
        int i, j, central;
        double pivote;
        central = (primero + ultimo) / 2;
        pivote = vector[central];
        i = primero;
        j = ultimo;
        do
        {
            while (vector[i] < pivote) i++;
            while (vector[j] > pivote) j--;
            if (i <= j)
            {
                int temp;
                temp = vector[i];
                vector[i] = vector[j];
                vector[j] = temp;
                i++;
                j--;
            }
        } while (i <= j);

        if (primero < j)
        {
            quicksort(vector, primero, j);
        }
        if (i < ultimo)
        {
            quicksort(vector, i, ultimo);
        }
    }

    private void mostrar()
    {
        Console.WriteLine("Vector ordenados en forma ascendente");
        for (int i = 0; i < h; i++)
        {
            Console.Write("{0} ", vector[i]);
        }
        Console.ReadLine();
    }
}

```

```

Metodo de Quick Sort
Cuantos longitud del vector: 8
ingrese valor 1: 6
ingrese valor 2: 3
ingrese valor 3: 2
ingrese valor 4: 7
ingrese valor 5: 1
ingrese valor 6: 0
ingrese valor 7: 65
ingrese valor 8: 100
Vector ordenados en forma ascendente
0 1 2 3 6 7 65 100

```

Resultado del código

Método De Ordenamiento Shell Sort

El ShellSort es una mejora del método de inserción directa que se utiliza cuando el número de elementos a ordenar es grande. El método se denomina “shell” en honor de su inventor Donald Shell y también método de inserción con incrementos decrecientes.

En el método de clasificación por inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es más pequeño - por ejemplo -, hay que ejecutar muchas comparaciones antes de colocarlo en su lugar definitivamente. Shell modifico los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con eso se conseguía la clasificación más rápida. El método se basa en fijar el tamaño de los saltos constantes, pero de más de una posición.

¿Cómo funciona?

1. Incrementos (gaps):

- Al inicio, se elige una secuencia decreciente de incrementos. Estos incrementos determinan la distancia entre los elementos que se compararán en cada pasada.
- Una secuencia común es usar una secuencia de números impares decrecientes hasta llegar a 1. Por ejemplo: 7, 3, 1.

2. Ordenamiento por Inserción en Subarreglos:

- Para cada incremento:
 - Se dividen los elementos del arreglo en subarreglos, tomando elementos separados por el incremento actual.
 - Se ordena cada subarreglo utilizando el método de inserción.
- Al usar incrementos grandes al principio, se permiten que los elementos "grandes" se muevan rápidamente hacia sus posiciones aproximadas.

3. Reducción de Incrementos:

- A medida que se avanza, los incrementos se reducen. Esto hace que los subarreglos sean cada vez más pequeños y más ordenados.
- Finalmente, cuando el incremento es 1, se realiza un ordenamiento por inserción en todo el arreglo, garantizando así que el arreglo esté completamente ordenado.

Ejemplo en código c#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PruebaVector
{
    class PruebaVector
    {
        private int[] vector;

        public void Cargar()
        {
            Console.WriteLine("Metodo de Shell Sort");
            Console.Write("Cuantos longitud del vector:");
            string linea;
            linea = Console.ReadLine();
            int cant;
            cant = int.Parse(linea);
            vector = new int[cant];
            for (int f = 0; f < vector.Length; f++)
            {
                Console.Write("Ingrese elemento " + (f + 1) + ": ");
                linea = Console.ReadLine();
                vector[f] = int.Parse(linea);
            }
        }

        public void Shell()
        {
            int salto = 0;
            int sw = 0;
            int auxi = 0;
            int e = 0;
            salto = vector.Length / 2;
            while (salto > 0)
            {
                sw = 1;
                while (sw != 0)
                {
                    sw = 0;
                    e = 1;
                    while (e <= (vector.Length - salto))
                    {
                        if (vector[e - 1] > vector[(e - 1) + salto])
                        {
                            auxi = vector[(e - 1) + salto];
                            vector[(e - 1) + salto] = vector[e - 1];
                            vector[(e - 1)] = auxi;
                            sw = 1;
                        }
                        e++;
                    }
                    salto = salto / 2;
                }
            }
        }

        public void Imprimir()
        {
            Console.WriteLine("Vector ordenados en forma ascendente");
            for (int f = 0; f < vector.Length; f++)
            {
                Console.Write(vector[f] + " ");
            }
            Console.ReadKey();
        }

        static void Main(string[] args)
        {
            PruebaVector pv = new PruebaVector();
            pv.Cargar();
            pv.Shell();
            pv.Imprimir();
        }
    }
}
```



```
Metodo de Shell Sort
Cuantos longitud del vector:7
Ingrese elemento 1: 5
Ingrese elemento 2: 0
Ingrese elemento 3: 9
Ingrese elemento 4: 1
Ingrese elemento 5: 3
Ingrese elemento 6: 6
Ingrese elemento 7: 54
Vector ordenados en forma ascendente
0 1 3 5 6 9 54
```

Resultado del código

Método De Ordenamiento Radix

Ordenamiento Radix es un algoritmo de ordenación no comparativo. Este algoritmo evita las comparaciones insertando elementos en cubos de acuerdo con el radix (Radix/Base es el número de dígitos únicos utilizados para representar números. Por ejemplo, los números decimales tienen diez dígitos únicos). Ordena los elementos basándose en los dígitos de los elementos individuales. Realiza la ordenación por conteo de los dígitos desde el menos significativo hasta el más significativo. También se ha llamado ordenación en cubo o digital y es muy útil en máquinas paralelas.

Cómo funciona?

1. Identificación del dígito menos significativo:

- Se selecciona el dígito menos significativo de todos los números (por ejemplo, el dígito de las unidades).

2. Creación de buckets:

- Se crean un número de buckets igual al número de dígitos posibles (por ejemplo, 10 buckets para números decimales).

3. Distribución de elementos:

- Cada número se coloca en el bucket correspondiente a su dígito menos significativo.

4. Recolección de elementos:

- Los elementos se recolectan de los buckets en orden, desde el bucket 0 hasta el último.

5. Repetición para dígitos más significativos:

- Se repiten los pasos 2, 3 y 4 para el siguiente dígito más significativo, y así sucesivamente, hasta que se hayan considerado todos los dígitos.

Ejemplo en código c#

```
using System;

public class RadixSort
{
    public static void Sort(int[] arr)
    {
        int max = arr[0];
        for (int i = 1; i < arr.Length; i++)
            if (arr[i] > max)
                max = arr[i];

        // Encuentra el número máximo de dígitos
        int exp = 1;
        while (max / exp > 0)
        {
            countingSort(arr, exp);
            exp *= 10;
        }
    }
}
```

```
public static void countingSort(int[] arr, int exp)
{
    int n = arr.Length;
    int[] output = new int[n];
    int[] count = new int[10];

    // Inicializa el arreglo de conteo
    for (int i = 0; i < 10; i++)
        count[i] = 0;

    // Almacena el
    // cuenta de cada dígito
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Calcula las posiciones acumuladas
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Construye el arreglo de salida
    for (int i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copia el arreglo
    // de salida al arreglo original
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

public static void Main()
{
    int[] arr = { 170, 45, 75, 90, 802 };
    RadixSort.Sort(arr);
    Console.WriteLine("Arreglo ordenado:");
    foreach (int i in arr)
        Console.Write(i + " ");
}
```

45 75 90 170 802

Resultado del código anterior con datos ingresados en el programa

Algoritmos de búsqueda

Búsqueda secuencial

la búsqueda secuencial es un método para encontrar un valor objetivo dentro de una lista. Ésta comprueba secuencialmente cada elemento de la lista para el valor objetivo hasta que es encontrado o hasta que todos los elementos hayan sido comparados.

¿Cómo Funciona?

1. **Inicio en el primer elemento:** Se comienza comparando el elemento que se busca con el primer elemento de la lista.
2. **Comparación:** Si el elemento buscado es igual al elemento actual, se ha encontrado y la búsqueda termina.
3. **Avanzar al siguiente elemento:** Si los elementos no coinciden, se pasa al siguiente elemento de la lista y se repite el proceso de comparación.
4. **Finalización:** La búsqueda termina cuando se encuentra el elemento buscado o cuando se llega al final de la lista sin encontrarlo.

Ejemplo de un método de búsqueda secuencial

```
namespace HolaMundo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Humano ingresa el numero a buscar de la lista");
            int numeroBuscar=int.Parse(Console.ReadLine());
            int[] listaNumeros = new int[7];
            for (int i = 0; i < 7; i++)
            {
                Console.WriteLine("Humano ingresa el elemento " + (i+1) + " de la lista");
                int numero = int.Parse(Console.ReadLine());
                listaNumeros[i] = numero;
            }
            int pos = 0;
            bool bandera = false;
            while (pos < 7 && bandera == false) {
                if (numeroBuscar == listaNumeros[pos]) {
                    bandera = true;
                    break;
                }
                pos++;
            }
            if (bandera == true)
            {
                Console.WriteLine("Felicidades humano se encontro el numero " + numeroBuscar + " en la posicion " + (pos+1));
            }
            else {
                Console.WriteLine("Lo siento humano no se encontro el numero "+numeroBuscar);
            }
            Console.Read();
        }
    }
}
```

```
Humano ingresa el elemento 1 de la lista
10
Humano ingresa el elemento 2 de la lista
454
Humano ingresa el elemento 3 de la lista
7987895
Humano ingresa el elemento 4 de la lista
211
Humano ingresa el elemento 5 de la lista
19
Humano ingresa el elemento 6 de la lista
545454
Humano ingresa el elemento 7 de la lista
87
Humano ingresa el numero a buscar de la lista
19
Felicidades humano se encontro el numero 19 en la posicion 5
```

Ejemplo del código ejecutado

Búsqueda Binaria

La búsqueda binaria es un algoritmo de búsqueda eficiente que se utiliza para encontrar la posición de un elemento específico dentro de una lista **ordenada**. A diferencia de la búsqueda secuencial, que examina cada elemento uno por uno, la búsqueda binaria divide la lista en dos partes repetidamente hasta encontrar el elemento buscado o determinar que no está presente.

El algoritmo de búsqueda binaria es el siguiente:

1. Se declaran los índices superior e inferior. El inferior en 0 y el superior con el tamaño del arreglo menos 1.
2. Se calcula el centro del arreglo con la siguiente formula: $\text{centro} = (\text{superior} + \text{inferior}) / 2$
3. Verificamos si el arreglo en la posición centro es igual al dato que buscamos. Si es igual significa que encontramos el dato y retornamos centro.
4. Si son diferentes verificamos si el arreglo en la posición centro es mayor al dato que queremos buscar. Si es mayor actualizamos superior: $\text{superior} = \text{centro} - 1$, si no actualizamos inferior: $\text{inferior} = \text{centro} + 1$.

Ejemplo en código

```
namespace BusquedaBinaria
{
    class Busqueda
    {
        private int[] vector;

        public void Cargar()
        {
            Console.WriteLine("Busqueda Binaria");
            Console.WriteLine("Ingrese 10 Elementos");
            string linea;
            vector = new int[10];
            for (int f = 0; f < vector.Length; f++)
            {
                Console.Write("Ingrese elemento " + (f + 1) + ": ");
                linea = Console.ReadLine();
                vector[f] = int.Parse(linea);
            }
        }

        public void busqueda(int num)
        {
            int l = 0, h = 9;
            int m = 0;
            bool found = false;

            while (l <= h && found == false)
            {
                m = (l + h) / 2;
                if (vector[m] == num)
                {
                    found = true;
                }
                if (vector[m] > num)
                {
                    h = m - 1;
                }
                else
                {
                    l = m + 1;
                }
            }
            if (found == false)
            {
                Console.WriteLine("\nEl elemento {0} no esta en el arreglo", num);
            }
            else
            {
                Console.WriteLine("\nEl elemento {0} esta en la posicion: {1}", num, m + 1);
            }
        }

        public void Imprimir()
        {
            for (int f = 0; f < vector.Length; f++)
            {
                Console.Write(vector[f] + " ");
            }
        }
    }

    static void Main(string[] args)
    {
        Busqueda pv = new Busqueda();
        pv.Cargar();
        pv.Imprimir();
        Console.WriteLine("\n\nElemento a buscar: ");
        int num = int.Parse(Console.ReadLine());
        pv.busqueda(num);
        Console.ReadKey();
    }
}
```

Resultado de la ejecución del código

```
Busqueda Binaria
Ingrese 10 Elementos
Ingrese elemento 1: 2
Ingrese elemento 2: 3
Ingrese elemento 3: 4
Ingrese elemento 4: 7
Ingrese elemento 5: 6
Ingrese elemento 6: 5
Ingrese elemento 7: 4
Ingrese elemento 8: 3
Ingrese elemento 9: 7
Ingrese elemento 10: 6
2 3 4 7 6 5 4 3 7 6

Elemento a buscar: 4
El elemento 4 esta en la posicion: 3
```

Búsqueda por función hash

La búsqueda por función hash es una técnica de búsqueda muy eficiente que se utiliza para encontrar elementos en una colección de datos. En lugar de buscar secuencialmente o utilizando una búsqueda binaria, una función hash asigna cada elemento a una ubicación específica en una estructura de datos llamada tabla hash. Esta ubicación se calcula aplicando una función matemática al valor del elemento.

¿Cómo funciona en C#?

En C#, las tablas hash se implementan principalmente a través de las clases `Dictionary<TKey, TValue>` y `HashSet<T>`.

- **Dictionary<TKey, TValue>**: Almacena pares clave-valor. La clave se utiliza para calcular el índice en la tabla hash, y el valor asociado se almacena en esa ubicación.
- **HashSet<T>**: Almacena solo valores únicos. El valor en sí se utiliza como clave para calcular el índice en la tabla hash.

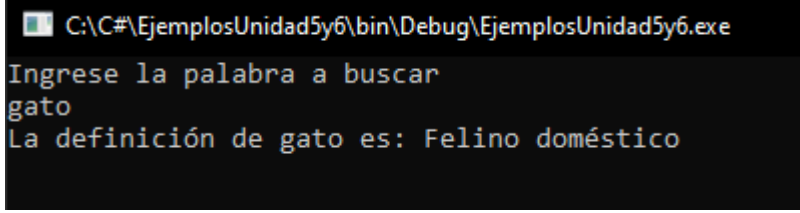
Proceso Básico:

1. **Función Hash:** Se aplica una función hash a la clave (o al valor en el caso de `HashSet`) para obtener un índice.
2. **Colisiones:** Si dos claves diferentes generan el mismo índice (colisión), se utilizan técnicas como encadenamiento o direccionamiento abierto para resolverlas.
3. **Acceso:** Para buscar un elemento, se aplica la misma función hash a la clave y se accede directamente a la ubicación calculada.

Ejemplo en código

```
// buscar una palabra
Console.WriteLine("Ingrese la palabra a buscar");
string palabraBuscar = Console.ReadLine();
if (diccionario.ContainsKey(palabraBuscar))
{
    Console.WriteLine($"La definición de {palabraBuscar} es: {diccionario[palabraBuscar]}");
}
else
{
    Console.WriteLine("La palabra no se encontró.");
}
Console.ReadLine();
```

Resultado del código ejecutado



```
C:\C#\EjemplosUnidad5y6\bin\Debug\EjemplosUnidad5y6.exe
Ingrese la palabra a buscar
gato
La definición de gato es: Felino doméstico
```

Conclusión

La elección del algoritmo de ordenamiento o búsqueda adecuado depende de varios factores, como el tamaño de los datos, la naturaleza de los datos (ordenados o desordenados), la frecuencia de las operaciones de búsqueda y los requisitos de espacio. Algoritmos como QuickSort y búsqueda binaria son generalmente preferidos por su eficiencia, mientras que otros como burbuja pueden ser útiles para pequeñas listas o en situaciones donde la simplicidad es más importante.

Bibliografía

- <https://csharp-facilito.blogspot.com/2013/07/metodo-de-ordenamiento-burbuja-en-c-sharp.html>
- https://aprendeprogramando.es/cursos-online/estructura_datos_csharp/busqueda_hash/
- <https://csharp-facilito.blogspot.com/2013/07/metodo-de-ordenamiento-shell-sort-en-c-sharp.html>
- https://www.ecured.cu/Ordenamiento_Radix