

Manejo de la vida adulta

**Sofia Vargas, Jorge Alandete,
Jeronimo Bermudez, Nicolas Quezada**

Clase grafo

agregarNodo(T nodo):

- Este método añade un nodo al grafo. Utiliza `putIfAbsent` para asegurarse de que el nodo se agrega solo si no existe previamente en la lista de adyacencia.

agregarArista(T nodo1, T nodo2):

- Añade una arista entre dos nodos `nodo1` y `nodo2`.

obtenerRelacionados(T nodo):

- Devuelve una lista de los nodos adyacentes de un nodo dado.

mostrarGrafo():

- Imprime todas las conexiones del grafo, mostrando cada nodo y su lista de nodos adyacentes.

mostrarRelacionesSimplificadas():

- Este método está diseñado para trabajar específicamente con objetos de tipo Tarjeta.

```
public class Grafo<T> {  
  
    private Map<T, List<T>> adjList = new HashMap<>();  
  
    // Añadir un nodo al grafo  
    public void agregarNodo(T nodo) {  
        adjList.putIfAbsent(key:nodo, new ArrayList<>());  
    }  
  
    // Añadir una arista entre dos nodos  
    public void agregarArista(T nodol, T nodo2) {  
        // Asegurarse de que ambos nodos existan en la lista de adyacencia  
        adjList.putIfAbsent(key:nodol, new ArrayList<>());  
        adjList.putIfAbsent(key:nodo2, new ArrayList<>());  
  
        // Ahora agregar la arista  
        adjList.get(key:nodol).add(e: nodo2);  
        adjList.get(key:nodo2).add(e: nodol);  
    }  
  
    // Obtener las tarjetas relacionadas con un nodo (vecinos)  
    public List<T> obtenerRelacionados(T nodo) {  
        return adjList.get(key:nodo);  
    }  
  
    // Visualizar todas las conexiones del grafo  
    public void mostrarGrafo() {  
        for (T nodo : adjList.keySet()) {  
            System.out.println("Tarjeta: " + nodo.toString() + " -> " + adjList.get(key:nodo));  
        }  
    }  
  
    public void mostrarRelacionesSimplificadas() {  
        for (T nodo : adjList.keySet()) {  
            System.out.println("Tarjeta: " + ((Tarjeta) nodo).resumen());  
            System.out.println("Relacionada con:");  
            for (T adyacente : adjList.get(key:nodo)) {  
                System.out.println(" - " + ((Tarjeta) adyacente).resumen());  
            }  
        }  
    }  
}
```

```

// Método para mostrar relaciones por una etiqueta dada
public static void mostrarRelacionesPorEtiqueta(String etiqueta, Lista<Tarjeta> lista, ConjuntoDisyunto<Tarjeta> conjuntoDisyunto) {
    Grafo<Tarjeta> grafo = new Grafo<>();

    // Construimos el grafo solo con las tarjetas que tienen la etiqueta dada
    Iterator<Tarjeta> iterador = lista.iterator();
    while (iterador.hasNext()) {
        Tarjeta tarjeta = iterador.next();
        if (tarjeta.getTag().equalsIgnoreCase(anotherString: etiqueta)) {
            // Agregar nodos y relaciones al grafo
            for (Tarjeta otraTarjeta : lista) {
                if (!tarjeta.equals(obj: otraTarjeta) && otraTarjeta.getTag().equalsIgnoreCase(anotherString: etiqueta)) {
                    grafo.agregarArista(nodo1: tarjeta, nodo2: otraTarjeta);
                }
            }
        }
    }

    // Mostrar las relaciones entre tarjetas con la etiqueta dada
    System.out.println("Mostrando relaciones entre tarjetas con la etiqueta: " + etiqueta);
    grafo.mostrarRelacionesSimplificadas();
}

```

Nueva funcion

- Tiene como objetivo mostrar las relaciones entre tarjetas que comparten una etiqueta específica dentro de una lista de tarjetas.
- Si la tarjeta tiene la etiqueta que se busca, se itera de nuevo sobre toda la lista para encontrar otras tarjetas que también tengan esa misma etiqueta.
- Se comprueba que las dos tarjetas sean diferentes y que la otra tarjeta también tenga la misma etiqueta.
- Se agrega una arista entre las dos tarjetas en el grafo. Esto significa que ambas tarjetas están relacionadas debido a que comparten la misma etiqueta.

```

// Cargar tarjetas
Lista<Tarjeta> lista = cargarTarjetas();
ConjuntoDisyunto<Tarjeta> conjuntoDisyunto = new ConjuntoDisyunto<>();
System.out.println("Ingrese la etiqueta por la que quiere ver las relaciones:");
String etiquetaIngresada = scanner.nextLine();
System.out.println("----- Comparacion de rendimiento sin grafo -----");
// Limpia la memoria antes de empezar a medir
runtime.gc();
long startMemory2 = runtime.totalMemory() - runtime.freeMemory(); // Memoria usada antes de la ejecución
long startTime2 = System.nanoTime();
// Ejecuta el método sin grafo
contarYMostrarRelacionesPorEtiqueta(etiqueta:etiquetaIngresada, lista);
long endTime2 = System.nanoTime();
long endMemory2 = runtime.totalMemory() - runtime.freeMemory(); // Memoria usada después de la ejecución
System.out.println("----- Comparacion de rendimiento con grafo -----");
// Limpia la memoria antes de empezar a medir
runtime.gc();
long startMemory1 = runtime.totalMemory() - runtime.freeMemory(); // Memoria usada antes de la ejecución
long startTimel = System.nanoTime();
// Ejecuta el método con grafo
mostrarRelacionesPorEtiqueta(etiqueta:etiquetaIngresada, lista, conjuntoDisyunto);
long endTime1 = System.nanoTime();
long endMemory1 = runtime.totalMemory() - runtime.freeMemory(); // Memoria usada después de la ejecución
System.out.println("Tiempo de ejecucion (relaciones por etiqueta sin usar grafo): " + (endTime2 - startTime2) + " nanosegundos.");
System.out.println("Memoria usada (relaciones por etiqueta sin usar grafo): " + (endMemory2 - startMemory2) + " bytes.");
System.out.println("Tiempo de ejecucion (relaciones por etiqueta usando grafo): " + (endTime1 - startTimel) + " nanosegundos.");
System.out.println("Memoria usada (relaciones por etiqueta usando grafo): " + (endMemory1 - startMemory1) + " bytes.");

```

Este código realiza una prueba de estrés (stressTest) para comparar el rendimiento de dos enfoques diferentes al mostrar relaciones entre tarjetas que comparten una etiqueta: uno sin usar un grafo y otro usando un grafo.

Stress test

```
public static void contarYMostrarRelacionesPorEtiqueta(String etiqueta, Lista<Tarjeta> lista) {
    // Contador de tarjetas con la etiqueta ingresada
    int count = 0;
    // HashMap para almacenar las relaciones de las tarjetas con la misma etiqueta
    HashMap<Tarjeta, Lista<Tarjeta>> relaciones = new HashMap<>();
    // Recorrer la lista de tarjetas y encontrar todas las que tengan la etiqueta ingresada
    for (Tarjeta tarjeta : lista) {
        if (tarjeta.getTag().equalsIgnoreCase(anotherString: etiqueta)) {
            count++;
            // Crear una lista para las relaciones de esta tarjeta
            Lista<Tarjeta> relacionadas = new Lista<>();
            // Buscar las relaciones con otras tarjetas que también tengan la misma etiqueta
            for (Tarjeta otraTarjeta : lista) {
                if (!tarjeta.equals(obj: otraTarjeta) && otraTarjeta.getTag().equalsIgnoreCase(anotherString: etiqueta)) {
                    relacionadas.add(data: otraTarjeta); // Añadir la tarjeta relacionada
                }
            }
            // Guardar las relaciones en el HashMap
            relaciones.put(key: tarjeta, value: relacionadas);
        }
    }
    // Mostrar el número total de tarjetas con la etiqueta ingresada
    System.out.println("Número de tarjetas con la etiqueta '" + etiqueta + "'": " + count);
    // Mostrar las relaciones
    System.out.println("Mostrando relaciones entre tarjetas con la etiqueta: " + etiqueta);
    for (Map.Entry<Tarjeta, Lista<Tarjeta>> entry : relaciones.entrySet()) {
        Tarjeta tarjeta = entry.getKey();
        Lista<Tarjeta> relacionadas = entry.getValue();

        System.out.println("Tarjeta: " + tarjeta.getTitle() + " esta relacionada con:");
        if (relacionadas.isEmpty()) {
            System.out.println(" Ninguna otra tarjeta.");
        } else {
            for (Tarjeta relacionada : relacionadas) {
                System.out.println(" - " + relacionada.getTitle());
            }
        }
    }
}
```

```
public static void mostrarRelacionesPorEtiqueta(String etiqueta, Lista<Tarjeta> lista, ConjuntoDisyunto<Tarjeta> conjuntoDisyunto) {
    Grafo<Tarjeta> grafo = new Grafo<>();

    // Construimos el grafo solo con las tarjetas que tienen la etiqueta dada
    Iterator<Tarjeta> iterador = lista.iterator();
    while (iterador.hasNext()) {
        Tarjeta tarjeta = iterador.next();
        if (tarjeta.getTag().equalsIgnoreCase(anotherString: etiqueta)) {
            // Agregar nodos y relaciones al grafo
            for (Tarjeta otraTarjeta : lista) {
                if (!tarjeta.equals(obj: otraTarjeta) && otraTarjeta.getTag().equalsIgnoreCase(anotherString: etiqueta)) {
                    grafo.agregarArista(nodo1: tarjeta, nodo2: otraTarjeta);
                }
            }
        }
    }

    // Mostrar las relaciones entre tarjetas con la etiqueta dada
    System.out.println("Mostrando relaciones entre tarjetas con la etiqueta: " + etiqueta);
    grafo.mostrarRelacionesSimplificadas();
}
```

Usando grafo			
Numero de tarjetas	Memoria (bytes)	Nanosegundos	Segundos
10	504,280	17,338,100	0.0173
20	629,280	70,307,900	0.0703
40	1,133,072	319,049,200	0.3190
80	1,892,280	1,034,996,400	1.03
160	1,203,656	4,634,717,700	4.63
320	38,785,552	19,017,703,200	19.02
640	41,654,688	80,748,613,000	80.75
1,280	54,582,640	358,586,255,700	358.59

Sin usar grafo			
Numero de tarjetas	Memoria (bytes)	Nanosegundos	Segundos
10	1,384,208	24,230,000	0.0242
20	1,384,208	38,952,100	0.0390
40	1,845,632	104,422,100	0.1044
80	2,560,616	349,541,700	0.35
160	5,580,584	1,763,442,300	1.76
320	40,655,123	7,596,002,400	7.60
640	46,512,985	31,331,353,300	31.33
1,280	60,703,912	134,747,023,300	134.75

Tiempo de ejecucion (relaciones por etiqueta sin usar grafo): 24230000 nanosegundos.
 Memoria usada (relaciones por etiqueta sin usar grafo): 1384208 bytes.
 Tiempo de ejecucion (relaciones por etiqueta usando grafo): 17338100 nanosegundos.
 Memoria usada (relaciones por etiqueta usando grafo): 504280 bytes.

Funcionalidad de Búsqueda

Búsqueda mediante `String.contains(T,P)`:

Retorna True si el patrón P está en texto T, en caso contrario retorna False.

`String.contains(T,P)` hace un llamado a `String.IndexOf(T,P)`, que retorna todas las posiciones del texto T en donde se encuentre el patrón P. Entonces `String.IndexOf(T,P)` recorre todo el texto T en busca de P, verificando si es igual para toda subcadena S, es decir hace una búsqueda por fuerza bruta $O(|T|^*|P|)$

Búsqueda por fuerza bruta

Toma una subcadena S de longitud $|P|$ del texto T

Compara carácter por carácter la subcadena S con la cadena patrón a buscar P $O(|P|)$

- Si la subcadena S es igual a la cadena patrón P retorna True
- Si la subcadena S NO es igual a la cadena patrón P , entonces toma la siguiente subcadena S de longitud $|P|$ dentro del texto T

Se repite hasta que se encuentre una subcadena S igual a el patrón P o hasta que se hayan comparado todas la subcadenas del texto $O(|T|-|P|)$

Comparar la subcadena S con la cadena patrón P tiene complejidad $O(|P|)$, este proceso en el peor de los casos lo hace un total de $|T|-|P|$ veces, por lo que la búsqueda por fuerza bruta tiene una complejidad computacional de $O(|T|*|P|)$

Búsqueda por Hash (Algoritmo Rabin-Karp)

Calcula el valor Hash de la cadena patrón P o la cadena a Buscar $O(|P|)$

Precomputa el valor Hash los caracteres del texto T $O(|T|)$

Calcula el valor Hash de la primera subcadena S_k del texto T $O(|P|)$

Compara si el hash de la subcadena S y el hash de la cadena patrón son iguales:

↳ Compara si la subcadena S_k y la cadena patrón son iguales:

- Si son iguales, entonces se encontró el patrón P en el texto T, retornando True. $O(q|P|)$ (con q siendo las veces en los que los Hash son iguales)
- Si no son iguales, entonces se calcula el valor Hash de la siguiente subcadena S_{k+1} mediante la actual subcadena S_k y el valor Hash del carácter nuevo de la cadena.

Se repite hasta que se encuentre una subcadena S igual a el patrón P o hasta que se hayan comparado todas las subcadenas del texto $O(|T|-|P|)$

Tiene $O(|T|*|P|)$ pero...

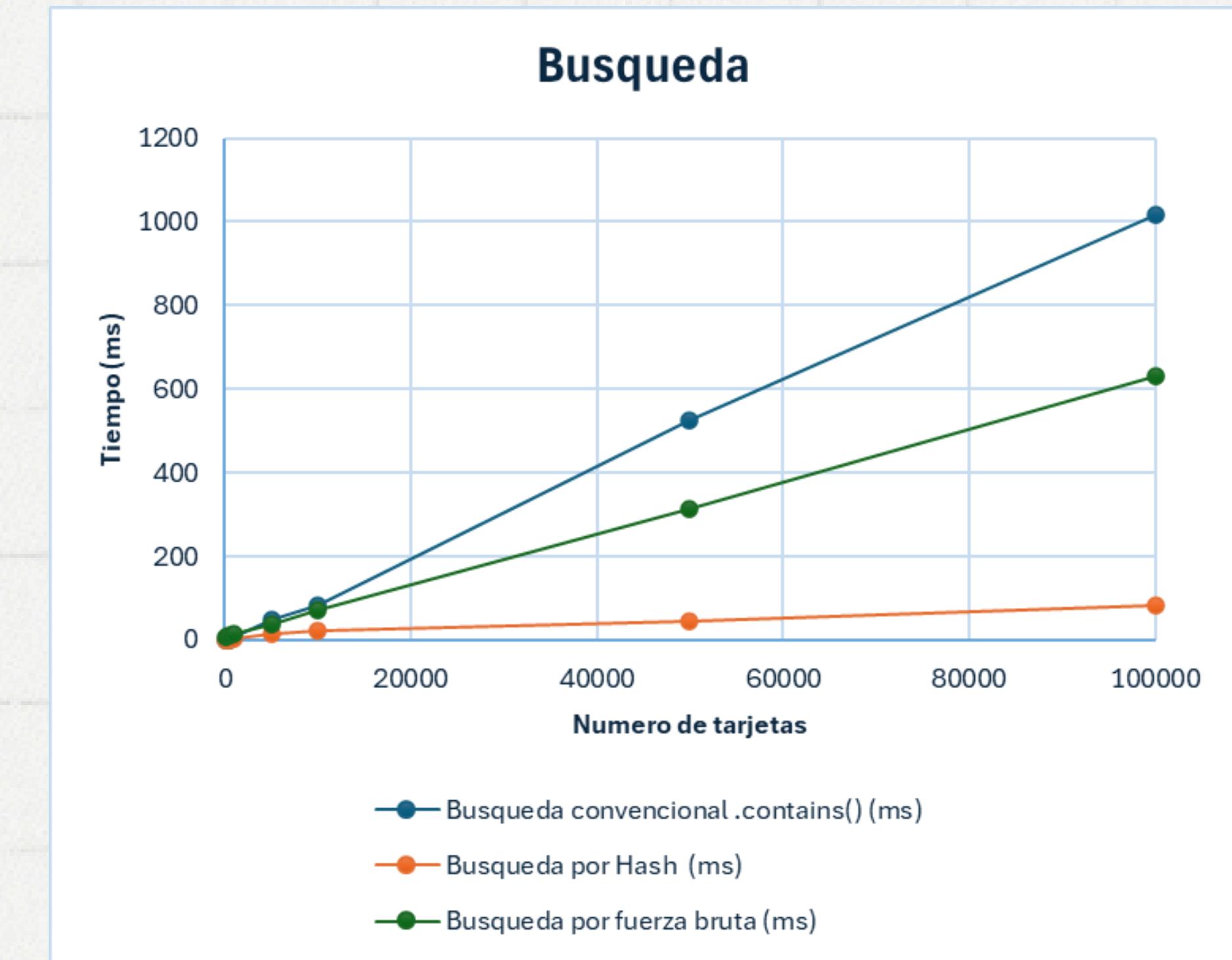
pase a tener una complejidad computacional de
 $O(|T| * |P|)$ (peor de los casos)

Al tomarse una buena función Hash (con un numero primo lo suficientemente grande) se evitan falsas colisiones en la función Hash, por lo que la complejidad computacional promedio al buscar un patrón **P** en un texto **T** se vuelve **$O(|T| + (q+1)|P|)$** donde **q** es el numero de colisiones de la función Hash.

por esto de importante elegir una buena función Hash en la que se eviten las colisiones.

Comparación búsqueda método .contains(), Hash y fuerza bruta

Numero de tarjetas	Numero de caracteres	Búsqueda .contains() (ms)	Búsqueda por Hash (ms)	Busqueda por fuerza bruta (ms)
100	12766	1	1	8
500	63831	1	1	12
1000	127660	8	4	15
5000	638300	47	13	39
10000	1276600	82	24	72
50000	6383000	525	46	314
100000	12766000	1017	82	630





Demosturacion del software final!