2017

.

TRABAJO PRÁCTICO

PROGRAMACION III

CIFUENTES/VÁZQUEZ

ETAPA 2

FACULTAD DE CIENCIAS
EXACTAS
UNIVERSIDAD NACIONAL DEL CENTRO
DE LA PROVINCIA DE BUENOS AIRES

Introducción

1 USUARIOS: ORDENAMIENTOS Y BUSQUEDAS MAS EFICIENTES

Etapa 2. Eficiencia Computacional

Hasta ahora, para verificar si existe un usuario debemos recorrer toda la estructura. En el peor caso, el tiempo insumido es proporcional a la cantidad de usuarios, o sea es O(n).

Se pide reducir la complejidad computacional (el tiempo insumido) de esta operación al orden logarítmico (O(log n)) mediante la implementación de algún método que ordene la estructura, y otro de búsqueda logarítmica (búsqueda binaria u otra técnica que crea conveniente).

Seleccione la estructura de representación de los usuarios y una estrategia que optimice la operación de verificación de existencia (búsqueda). Implemente estas operaciones siguiendo la propuesta elegida y justifique.

Resolución

2 EN BREVE

Tomando en cuenta lo implementado en la primera etapa, elegimos continuar la etapa 2 en la estructura de arreglo utilizando como método de búsqueda el **Binary Search** y el **QuickSort** como método de búsqueda, lo cual agregó funcionalidades al paquete “arreglo”.

3 CLASES Y SU IMPLEMENTACIÓN

PAQUETE ARREGLO

1. Usuario.java
   Compuesto por:

   Atributos:
   - **DNI** de tipo String
   - **Gustos** de tipo String

Métodos:

- **getGustos**: encargada de devolver el arreglo de gustos
- **getDni**: la cual devuelve el DNI
- **generarGustos**(String[] datos): función encarga de generar el arreglo de gustos a partir del cvs y de guardarlos en atributo gustos
- **contieneGusto:** verifica si el arreglo de gustos ya contiene ciertos gustos para evitar repeticiones.

2. ArregloUsuario.java
   Compuesto por:

   Atributos:
   - **Arreglo** de tipo Usuario [ ]
   - **cantElem** de tipo int (con un valor inicial de 0)
   - **ordenado** de tipo boolean (valor inicial false)

   Métodos:

   - **agregar** (Usuario elem): Procedimiento que se encarga de cargar un elemento dado en el arreglo de String y de aumentar el tamaño del arreglo en caso de que no entren más elementos, además de modificar la variable “ordenado” a false, informando que el arreglo esta desordenado.
   - **contiene** (String aux): Función que devuelve un booleano luego de realizar la búsqueda de un elemento dado en el arreglo.
   - **eliminarAt** (int pos): Elimina el elemento que se encuentre en la posición dada.
   - **insertarAt** (int pos, String elem): agrega un elemento en una posición dada del arreglo.
   - **getCantElem**: devuelve la cantidad de elementos que hay actualmente en el arreglo.
   - **ordenarArregloQuickSort** método implementado para ordenar el arreglo
   - **existe** este método es un buscador binario, con el cual se realizan las búsquedas en el arreglo, también modifica la variable “ordenado” a true informando que el arreglo esta ordenado.

3. Quicksort.java
   Compuesto por:

   Atributos:
   - **array** de tipo Usuario []

   Métodos:
   - **sort y partición**, encargados de que, al recibir un arreglo, lo ordenen utilizando el método de Quicksort.
4. Simulador/2/3.java
   Compuesto por:

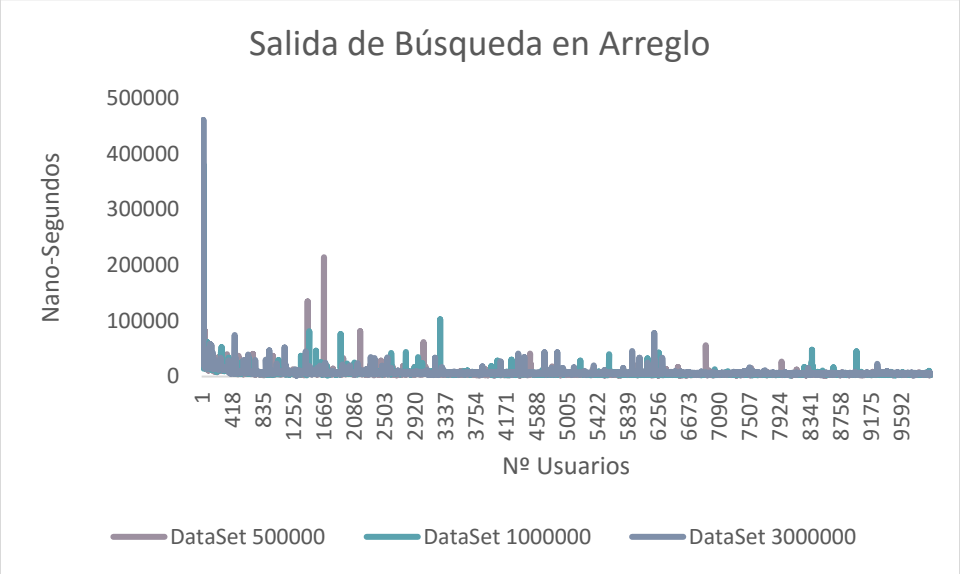
   Atributos:
   - **Usuarios** de tipo ArregloUsuario

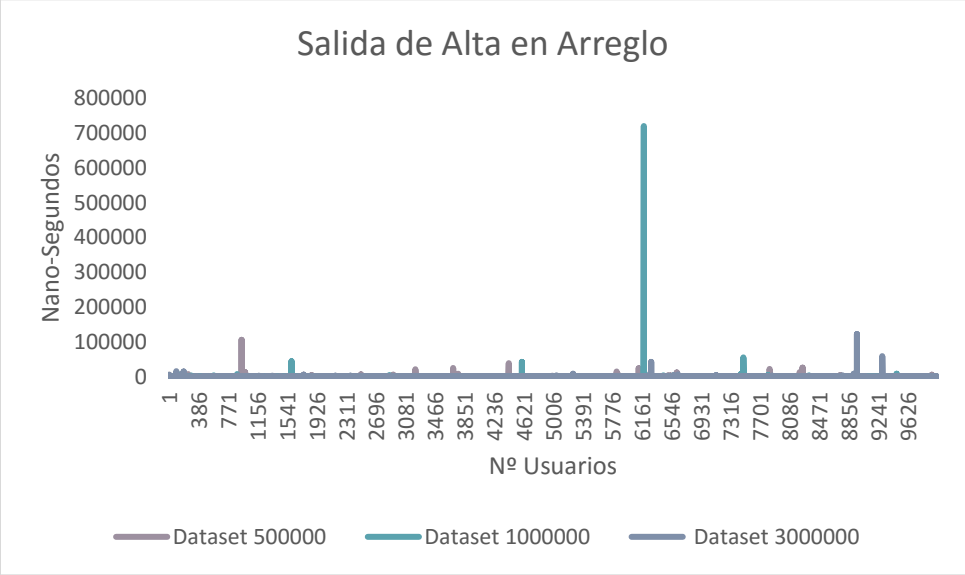
   Métodos:
   - **Reader**: toma la información de los distintos csv y las vuelca en el arreglo de **Usuarios.**
   - **writterBusqueda**: toma los datos del archivo busqueda.csv y los busca en el arreglo **Usuarios,** generando el archivo salidaBusqueda.csv**.**
   - **writterAltas:** toma los datos del archivo insert.csv y los inserta en el arreglo de **Usuarios,** generando el archivo salidaAltas.csv**.**
   - **main:** inicializa el arreglo de **Usuarios** y llama a las funciones anteriormente dichas.

4 GRÁFICOS:

En el gráfico de Salida de búsquedas, eliminamos el primer elemento buscado debido a que el tiempo transcurrido era demasiado elevado y hacia muy dificultosa la percepción del gráfico.

En el gráfico de Salida de Altas los tiempos son similares salvo en un valor, al que suponemos que se debe a un salto en la memoria.


Salida de Búsqueda en Arreglo
Nano-Segundos
500000
400000
300000
200000
100000
0
1
418
835
1252
1669
2086
2503
2920
3337
3754
4171
4588
5005
5422
5839
6256
6673
7090
7507
7924
8341
8758
9175
9592
Nº Usuarios
DataSet 500000
DataSet 1000000
DataSet 3000000


Salida de Alta en Arreglo
Nano-Segundos
800000
700000
600000
500000
400000
300000
200000
100000
0
1
386
771
1156
1541
1926
2311
2696
3081
3466
3851
4236
4621
5006
5391
5776
6161
6546
6931
7316
7701
8086
8471
8856
9241
9626
Nº Usuarios
Dataset 500000
Dataset 1000000
Dataset 3000000

5 TIEMPOS TOTALES Y PROMEDIOS

Los tiempos de las tablas están en nano-segundos.

Tiempo Altas: Total/Promedio	Altas	Búsqueda
500000	6253496/625,3496	644851418/64485,1418
1000000	6503653/650,3653	1401708731/140170,8731
3000000	5593503/559,3503	4749240845/474924,5285

En estas tablas mostramos los tiempos totales y el promedio de las altas y búsquedas de los distintos archivos generados. Al buscar los peores tiempos de cada archivo, descubrimos que esos tiempos eran generados por un salto en la memoria cache, por lo cual decidimos omitirlos, ya que el promedio de iteraciones era mucho menor.

Conclusión

Con la solución que implementamos, pensamos que abarcamos los 2 escenarios propuestos, ya que si sucediera el escenario a) se llevarían a cabo las altas y cuando se quisiera realizar las búsquedas con solo un ordenamiento, el arreglo ya estaría listo para realizarlas. En el caso del escenario b) se pueden realizar todas las altas que se requieran, y luego al realizar las pocas búsquedas, el arreglo se ordenaría una sola vez.

En el caso a) se podría haber implementado un árbol binario de búsqueda, lo cual mejoraría el tiempo de las inserciones y las verificaciones de existencia, pero aumentaría la complejidad temporal y el mantenimiento en base al trabajo que estamos realizando. Por lo tanto, no sería conveniente implementarlo, ya que los tiempos variarían muy poco, comparados con nuestra solución basada en un arreglo.

En ciertos tipos de árboles se implementan rotaciones a izquierda o derecha para mantener el balance después de una inserción o eliminación.