



Ciencias de la
Computación
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

Tarea 4

CC3001 – Algoritmos y Estructuras de Datos

Nombre:	Sebastián Parra
Profesor:	Patricio Poblete
Auxiliares:	Gabriel Flores
	Sven Reisenegger
Ayudantes:	Fabián Mosso
	Gabriel Chandía
	Matías Risco
	Matías Ramírez

Fecha: 16 de julio de 2018

Introducción

El siguiente documento trata sobre la tarea 4 del curso CC3001 – Algoritmos y Estructuras de Datos. En éste se describe brevemente el problema que se tuvo que solucionar, luego se procede a analizar el problema de manera detallada, reconociendo los supuestos y casos de borde que se debe considerar, para posteriormente explicar cómo funciona la solución implementada, cómo se obtuvo, y cuáles son las indicaciones principales que se debe saber para poder ejecutar correctamente el programa. Finalmente, se exponen los resultados obtenidos, se analizan, y se extraen conclusiones.

La tarea consistió en implementar en java una estructura de árbol cartesiano, que es una especie de árbol binario en donde cada nodo almacena un par ordenado (x, y) , cuyas coordenadas x forman un árbol de búsqueda binaria estándar, mientras que sus coordenadas y forman un árbol de prioridad tipo *heap*. Además, se solicitó implementar las funciones de inserción, impresión, y de cálculo del costo promedio de búsqueda, para luego realizar un análisis.

Para solucionar el problema, se implementaron tres clases fundamentales:

1. Tupla, para modelar los pares ordenados de los nodos
2. Pila, que se utilizó para realizar la operación de inserción
3. nodoLista, que modela una lista enlazada y se utilizó para construir Pilas
4. ArbolCartesiano, que es la clase principal del programa.

Luego, utilizando estas clases se implementaron las siguientes funciones:

1. `insertar(x, y)`, la cual funciona efectuando una inserción tipo ABB según la coordenada x de a tupla, seguida de la realización de operaciones de rotación simple hasta que el nodo quede correctamente ubicado en su coordenada y
2. `imprimir()`, la cual recorre el árbol en postorden e imprime sus nodos en notación polaca inversa
3. `costoPromedio()`, la cual calcula el costo promedio de búsqueda del árbol sumando los costos de búsqueda de todos los nodos internos, y dividiéndolo por el número de nodos internos.

Análisis del Problema

Como se planteó en la sección anterior, el objetivo de la tarea fue utilizar java para implementar un tipo de árbol cartesiano cuyas coordenadas en x fuesen números enteros y funcionasen como un árbol binario, mientras que sus coordenadas y debieron ser números de punto flotante que funcionasen como un *heap*. En cuanto a las funciones a implementar, es importante destacar que se solicitó un formato específico para la función de impresión del árbol cartesiano: Se debe recorrer el árbol en postorden e imprimir sus nodos de la forma " (x,y) ", utilizando el string "[]" para los nodos vacíos.

En cuanto a la segunda parte de la tarea, ésta consistió en realizar un análisis de los costos promedio de búsqueda en el árbol mediante el siguiente experimento:

- Elegir un valor $n=2^k$, donde k es un número entre 10 y 16
- Introducir todos los números del 1 al n en un árbol cartesiano vacío, donde las prioridades (coordenada y) de cada número fuesen elegidas al azar
- Repetir el experimento 10 veces para el mismo n , y para cada valor de k
- Obtener el costo promedio de búsqueda para todos los valores de n y analizar su comportamiento en función de $\log_2(n)$

Para el correcto funcionamiento del programa entregado, se debe tener en mente que se consideraron una serie de supuestos con respecto al problema. En primer lugar, no se consideran los casos en donde se desea insertar un nodo con alguna (o ambas) de las coordenadas iguales a las de un nodo existente, por lo que el comportamiento del código frente a estos casos no fue estudiado. En segundo lugar, se asumió que los archivos que leerá el programa tendrán sus entradas separadas por un salto de línea y seguirán un formato igual al de los ejemplos subidos en material docente. Es decir, los archivos tendrán la siguiente forma:

X1	Y1
X2	Y2
...	
Xn	Yn
EOF	

Finalmente, es importante mencionar también que la solución implementada es capaz de manejar el caso de borde de una entrada vacía (árbol vacío), imprimiendo correctamente el string “[]” y un costo de búsqueda de cero.

Solución del Problema

Para llegar a la solución final del problema, se fueron implementando las funciones pedidas una por una. Inicialmente, se crearon las clases `ArbolCartesiano` y `Tupla`, que fueron las primeras necesidades que se notaron antes de tener que implementar cualquier función. Luego, para la función de inserción de nodos, se notó que sería necesario “recordar” los nodos del árbol que la función ha recorrido para poder implementar las rotaciones simples al ordenar los nodos según su prioridad. Debido a esta necesidad, se decidió implementar las clases `nodoLista` y `Pila`, para ingresar el camino de nodos en una pila, y así poder saber cuáles son los padres del nodo, tanto inmediatamente después de haber sido ubicado según su coordenada x, como después de haber realizado una (o varias) rotaciones simples.

En cuanto a la función de impresión del árbol cartesiano, ésta se implementó mediante funciones recursivas, donde se reconoció la siguiente rutina:

- Si el nodo está vacío, imprimir el string “[]”
- Si no está vacío, imprimir recursivamente el subárbol izquierdo, luego el derecho, y finalmente imprimir el valor del nodo.

Finalmente, para el cálculo de costo promedio se utilizaron dos funciones recursivas: La primera se encarga de recorrer todos los nodos del árbol, notando que, si un nodo se encuentra vacío, el costo de búsqueda es cero, y si no, el costo será igual a la altura del nodo (o bien, la distancia entre la raíz y el nodo, más uno). Por otro lado, la segunda se encarga de calcular el número de nodos no vacíos del árbol, para lo cual realiza un recorrido por todos sus nodos, y cada vez que se encuentra con un nodo no vacío, suma 1 a un contador.

Teniendo estas tres funciones, se implementó una función principal que es capaz de solucionar ambas partes de la tarea, dependiendo de cómo se defina un parámetro del código.

Modo de Uso

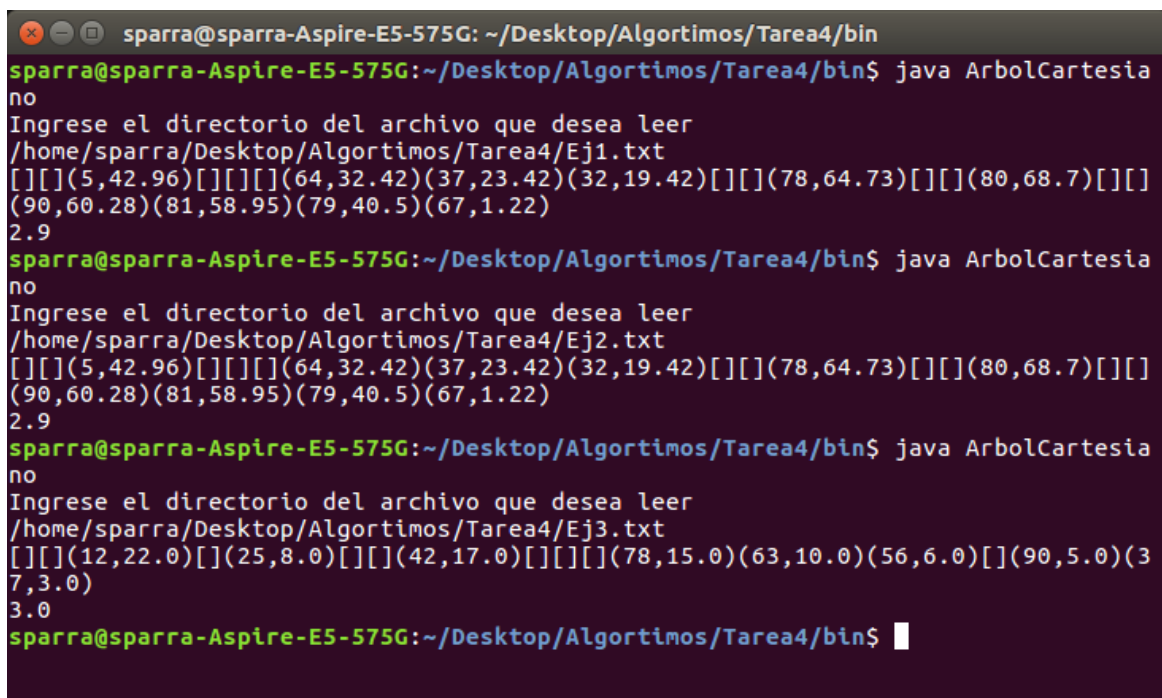
La ejecución del código es bastante simple, sólo necesitando ajustar una o dos variables, dependiendo de la parte de la tarea que se desee ejecutar.

En primer lugar, en la línea 206 se define una variable llamada `usarTreap`, que es un booleano que se debe fijar como `true` si se desea ejecutar la parte 2 de la tarea, o `false` si se desea ejecutar la primera parte. Si se fija como `false` y se ejecuta el programa, éste preguntará por el directorio del archivo que se desee ingresar como entrada, ante lo cual el usuario deberá ingresar el directorio completo del archivo (por ejemplo, `/home/Desktop/MiArchivo.txt`). Una vez ingresado el directorio, el programa se ejecutará y debería imprimir el árbol resultante con su costo promedio de búsqueda sin ser necesario hacer nada más. En cambio, si se fija el booleano como `true`, también será necesario cambiar el valor de `n` en la línea 241 del código, el cual tendrá que seguir la fórmula $n=2^k$, con `k` entre 10 y 16. Luego de cambiar estos dos parámetros, se puede ejecutar el programa y éste imprimirá los 10 valores obtenidos para el costo promedio de búsqueda.

Resultados

Parte 1

A continuación, en la figura 1 se presentan los resultados obtenidos en la primera parte para los tres ejemplos con solución que se encuentran en el archivo “ejemplos.txt” subido a material docente. El orden en que se introdujeron los ejemplos fue el mismo en el cual aparecen en el archivo.



```
sparra@sparra-Aspire-E5-575G: ~/Desktop/Algoritmos/Tarea4/bin
sparra@sparra-Aspire-E5-575G:~/Desktop/Algoritmos/Tarea4/bin$ java ArbolCartesia
no
Ingrese el directorio del archivo que desea leer
/home/sparra/Desktop/Algoritmos/Tarea4/Ej1.txt
[[[](5,42.96)[[][(64,32.42)(37,23.42)(32,19.42)[[](78,64.73)[[](80,68.7)[[]
(90,60.28)(81,58.95)(79,40.5)(67,1.22)
2.9
sparra@sparra-Aspire-E5-575G:~/Desktop/Algoritmos/Tarea4/bin$ java ArbolCartesia
no
Ingrese el directorio del archivo que desea leer
/home/sparra/Desktop/Algoritmos/Tarea4/Ej2.txt
[[[](5,42.96)[[][(64,32.42)(37,23.42)(32,19.42)[[](78,64.73)[[](80,68.7)[[]
(90,60.28)(81,58.95)(79,40.5)(67,1.22)
2.9
sparra@sparra-Aspire-E5-575G:~/Desktop/Algoritmos/Tarea4/bin$ java ArbolCartesia
no
Ingrese el directorio del archivo que desea leer
/home/sparra/Desktop/Algoritmos/Tarea4/Ej3.txt
[[[](12,22.0)[(25,8.0)[[][(42,17.0)[[][(78,15.0)(63,10.0)(56,6.0)[(90,5.0)(3
7,3.0)
3.0
sparra@sparra-Aspire-E5-575G:~/Desktop/Algoritmos/Tarea4/bin$
```

Figura 1: Resultados obtenidos para los ejemplos con solución del archivo "ejemplos.txt".

Como se puede notar en la figura, los resultados coinciden con las soluciones mostradas en el archivo, lo cual corrobora el correcto funcionamiento del programa, y permite poder avanzar a la siguiente parte de la tarea con mayor seguridad.

Parte 2

En el gráfico de la figura 2 se muestran los resultados obtenidos para el costo promedio de búsqueda versus n , donde n fue graficado en escala logarítmica. Además, en el gráfico también se muestra una curva de ajuste logarítmico a los puntos, junto con la ecuación que la modela.

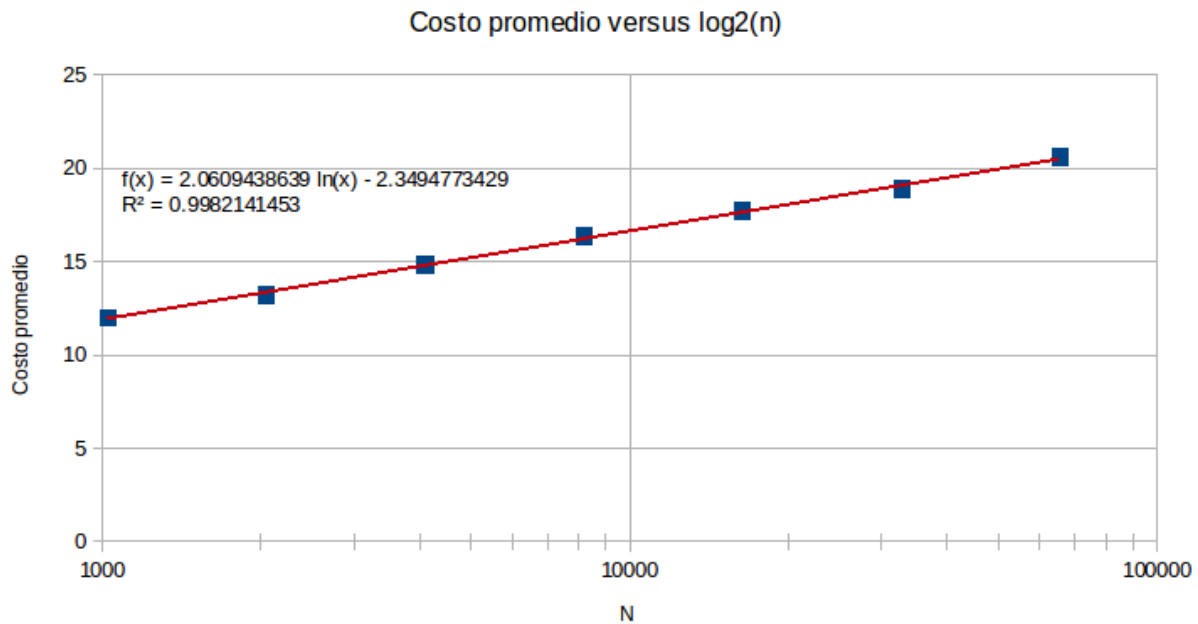


Figura 2: Gráfico costo promedio versus $\log_2(n)$ para los valores de n probados. El eje x se encuentra en escala logarítmica.

Como se puede notar al observar el gráfico de esta figura, la curva presenta un alto grado de ajuste a los puntos, lo cual corrobora la proporcionalidad que existe entre el costo de búsqueda en un *treap* versus el logaritmo de la cantidad de nodos, pudiendo así concluir que el costo de búsqueda es efectivamente $O(\log n)$ para este tipo de estructuras. Finalmente, si se observa la ecuación de la curva de ajuste, se puede determinar que el coeficiente que acompaña al logaritmo es aproximadamente 2.