Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Binary Tree v/s Splay Tree
Tarea 2 - Diseño y análisis de algoritmos

Integrantes: Francisco Almeida.
Patricio Espinoza A.
Scarlett Plaza.

Profesor: Gonzalo Navarro

Auxiliar: Máximo Flores V.
Sergio Rojas H.

Fecha de entrega: 6 de Septiembre de 2024

Índice

1. Introducción

El hashing lineal corresponde a un algoritmo de tabla de hash utilizado para hacer búsqueda de un objeto en tiempo O(1). En esta implementación en particular, se expande la tabla de hashing de manera gradual, es decir, cuando al realizar búsquedas el número máximo de accesos promedio permitidos es superado, se realiza una expansión a la siguiente página, leyéndola y reinsertando sus objetos.

En el siguiente informe se presentan los resultados obtenidos de la implementación del algoritmo de hashing lineal al realizar inserciones de elementos en memoria principal mediante el lenguaje Java y el paquete de gráficos JFreeChart. En este mismo marco, se explicará la estructura de clases dentro de nuestra implementación a fin de garantizar una reproducibilidad del experimento.

Para este estudio se analizan dos aspectos del hashing lineal al realizar inserciones para una cantidad de elementos; la variación del costo promedio real de inserción cuando se modifica la variable controlada $c_{\text{máx}}$, que representa el máximo costo promedio permitido, y la relación entre el porcentaje de llenado de las páginas y el costo promedio real. Para ello, cada acción de acceso, escritura o borrado de una página es considerado como una I/O.

La fase experimental es realizada mediante la inserción de N números enteros de 64 bits con $|N| \in \{2^{10}, 2^{11}, 2^{12}, ..., 2^{24}\}$, donde cada uno de estos casos se combina con los valores asignados de $c_{\text{máx}} \in \{10, 30, 50, 70, 90\}$.

Para este algoritmo el costo de inserción de hashing lineal es O(1), sin embargo, la implementación que se está utilizando no es la más eficiente debido a la metodología obligatorio de recorrer para verificar si un elemento está o no. Por lo tanto, se plantea como hipótesis que el rendimiento experimental será menos eficiente. Asimismo, se espera que a medida que se incremente el costo máximo promedio permitido los resultados del experimento se acerquen más al rendimiento óptimo del hashing lineal.

2. Desarrollo

Para la implementación del algoritmo de hashing lineal se utilizó el lenguaje Java debido a que es compilado y sigue siendo más rápido que lenguajes como Python. El lenguaje usado es orientado a objetos, y por tanto se construyeron una serie de clases enunciadas a continuación.

2.1. Clase Página

En esta clase se define la estructura de la página, la cual puede contener como máximo 128 elementos de tipo Long, el cual es el entero de 64 bits en Java. Esto debido a que cada página debe ser de 1024 bits, por lo contendría 1024/64=128 elementos cada página. Así, la clase contiene un arreglo para guardar los elementos y un contador de la cantidad de elementos totales en la página.

A continuación se mostrarán los diferentes métodos que contiene esta clase:

- **Pagina()**: Corresponde al constructor, inicializa la página vacía con espacio para 128 elementos y un contador de elementos en 0.
- **getPagina()**: Devuelve un arreglo con los elementos en la página.
- **estaLlena()**: Verifica si la página está llena o no, devolviendo true si lo está.
- **getNumElementos()**: Indica la cantidad de elementos que hay en esa página.
- **insertarElemento(long elemento)**: Inserta un nuevo elemento en la página si no está llena. Retornando true si la inserción es exitosa, y false si la página está llena.
- **obtenerElemento(int index)**: Recupera un elemento de una posición dada si el indice está en rango, de lo contrario lanza una excepción indicando que esta fuera de rango.
- **contieneElemento(long elemento)**: Revisa si el elemento dado se encuentra en la página recorriendo todos los elementos almacenados. Retorna true si esta y false si no fue encontrado.
- **imprimirElementos()**: Imprime todos los elementos actuales de la página.

2.2. Clase Hashing

Esta clase implementa el algoritmo desarrollado para el hashing lineal, para ello se implementa una estructura de hash basada en páginas que permite gestionar los datos, aplicando las inserciones y accesos mediante un sistema de expansión dinámica de la tabla de hash.

Las estructuras que contiene consisten en: la tabla de hash que almacena listas de páginas, el número de páginas actualmente en uso “p”, el tamaño de la tabla “t”, el contador de accesos “totalIOs”, el contador de inserciones realizadas “totalInserciones”, la variable controlada “maxCostoPromedio” y la suma total de los costos “sumaIOs”

El constructor toma el hiperparámetro del máximo costo promedio e inicializa la tabla de hash con una página y establece los parámetros p=1, t=0, además de los contadores de inserciones y accesos I/O.

Posteriormente, se implementan los siguientes métodos:

- **imprimirTabla()**: Imprime todo el contenido de la tabla de hash, mostrando los elementos de cada página.
- **getPromedio()**: Calcula el costo promedio de I/Os por cada inserción.
- **hashLong(long input)**: Corresponde a la función de hash que dado un valor de tipo long entrega el valor de esta, asegurando que estos estén bien distribuidos y sean deterministas positivos dentro del rango entre 1 y $2^{64} - 1$.

- **insertar(long y)**: Dado un elemento y, calcula su función de hash y lo inserta en la tabla de hash, para esto se obtiene el índice k que es donde debe ser insertado de la tabla mediante el módulo de 2^{t+1} para segurar que el indice esté dentro del rango de la tabla. Si k esta dentro de las páginas actuales de la tabla, se llama al método insertarEnPagina para insertarlo en la página correspondiente, en caso de que k esté más allá de las páginas actuales se inserta en la página $k - 2^t$ mediante el mismo método antes mencionado.
- **insertarEnPagina(int index, long y)**: Este método recibe un indice de la tabla de hash donde se quiere insertar el elemento "y", este comienza por crear un contador de I/Os verificando que el índice entregado este dentro de los rangos de la tabla de hash y en caso de que no esté lanza una excepción indicándolo. Luego accede a la lista de páginas de la tabla, lo que conlleva un acceso, por lo cual suma uno al contador de I/Os. Como último paso antes de hacer la inserción del elemento verifica si es que este ya se encuentra en la tabla, para esto recorre cada página y en caso de que ya se encuentre en ella no es necesaria la inserción, esto tiene un costo de un I/Os por cada página, lo cual es sumado al contador.

  Una vez hecho esto se comprobó que es necesario hacer la inserción del elemento, por lo cual se comienza por acceder a la última página de la lista pues es donde se realizará la inserción, acá nos enfrentamos a dos escenarios:

  - (1) Si la página no está llena se inserta el elemento aumentando en 1 el contador de I/Os ya que se escribe en la página, además, se actualizan los contadores de la clase para mantener los valores del costo de inserción de todos los elementos y de todas las inserciones que se han realizado.
  - (2) Por otra parte si la página se ha llenado se crea una nueva página de rebalse, se inserta el elemento ahí, se añade esta nueva página a la lista y se aumenta el contador de I/Os en 2, uno por crear la nueva página y otro por añadir el elemento a esta. Una vez hecho esto se actualizan los contadores de la clase mencionados en el caso anterior y se calcula el costo promedio de la inserción para evaluar si es necesario expandir la tabla de hashing usando el método expandirPagina.
- **insertarRehashing(int index, long y)**: Dado el índice de la tabla de hash donde se quiere insertar el elemento "y", se comienza por verificar que el índice este dentro del rango de la tabla de hash, de no estarlo manda una excepción. Luego obtiene la lista de páginas en la posición dada y las recorre para verificar si el elemento ya fue insertado, en cuyo caso no se reinserta. En caso contrario se obtiene la última página y se verifica si esta está llena, de no estarlo se inserta el elemento "y" ahí, por otro lado si la página esta llena se crea una nueva página donde el elemento es insertado y la página se inserta en la lista.
- **expandirPagina()**: Para hacer la expansión de página se comienza por calcular cuál debe ser expandida, luego se crea una nueva página inicialmente vacía y se añade a la tabla de hash aumentando su tamaño. Posteriormente se utiliza una lista para guardar los elementos que se van a reinsertar recorriendo las páginas que serán expandidas y guardando sus elementos ahí. A continuación se limpian las páginas que se expandirán y se agrega la página nueva en su lugar, sumando un I/Os en cada caso.

  Para reinsertar los elementos se recorre la lista donde habían sido copiados, se calcula su función de hashing y la posición en la tabla, utilizando el método insertarRehashing para hacer la reinserción. Finalmente se actualiza la cantidad de páginas y si se cumple que la cantidad de páginas alcanza $2^t + 1$, se aumenta el tamaño de la tabla de hashing.
- **getCantidadPaginas()**: Inicializa un contador del total de páginas en 0 y recorre la lista de páginas de la tabla de hash aumentando el contador.
- **getPorcentajeLlenado()**: Comienza declarando una variable para acumular el porcentaje de llenado de la tabla de hash y luego itera sobre esta. sumando la cantidad de elementos por página para cada posición de la tabla, en base a esto se calcula el porcentaje de llenado por página y eventualmente el

porcentaje de llenado de cada posición de la tabla, este último se divide por la cantidad de páginas en esa posición para obtener el promedio de cada índice. Finalmente, la suma de cada índice se divide por el tamaño de la tabla y se obtiene el porcentaje de llenado de esta. Cabe mencionar que se utiliza esta técnica para calcular el porcentaje de llenado pues nos entrega una visión más detallada de como se están llenando las páginas dando una mejor idea del estado real de cada entrada.

2.3. Clase Graficar

Esta corresponde a la clase donde se crean los gráficos que son de interés para el análisis de resultados, para esto se definen los siguientes métodos:

- **graficarCostoPromedioVsDatosMultipleCmax(List listaCantidadDatos, List listaCostosReales, double costosMaximos)**:

Se encarga de graficar el costo promedio con respecto a la cantidad de datos para cada valor de costo máximo.

- **graficarLlenadoVsCostos(List cantidadDatos, List porcentajesLlenado, List costosReales, double cmáx)**:

Este método se utiliza para graficar la relación entre el porcentaje de llenado y los costos reales promedios reales en función de la cantidad de datos.

2.4. Clase Experimentación

Esta clase se utiliza para hacer la experimentación de hashing lineal, para esto define una lista con los tamaños de la cantidad de elementos con los que se experimentará y los costos máximos que definimos para el costo promedio de inserción, en nuestro caso comenzamos desde 10 hasta 90, de 20 en 20. También se inicializa un archivo llamado resutados_experimentos.txt donde se guardarán los resultados obtenidos para todo el experimento. Las variables usadas consisten en las listas de cantidad de datos, costos reales, porcentajes de llenado y costos reales de llenado.

Para cada costo máximo se inicializan las primeras tres listas, luego se recorren los tamaños de datos para generar el experimento utilizando la clase Hashing antes descrita y se insertan los N datos aleatoriamente en la tabla de hash, recolectando datos cada 1000 inserciones. Finalmente se guarda en el archivo txt la información de la cantidad de datos insertados, junto a el costo promedio máximo utilizado, el costo promedio real y el porcentaje de llenado.

Finalmente, la información almacenada se utiliza para graficar el costo promedio con respecto a la cantidad de datos y la relación entre el promedio de llenado con respecto a los costos reales para cada costo máximo.

Resumen:
- Inserción de N números enteros de 64 bits con $|N| \in \{2^{10}, 2^{11}, 2^{12}, ..., 2^{24}\}$
- Variable controlada $c_{\text{máx}} \in \{10, 30, 50, 70, 90\}$.

3. Resultados

Para la realización del experimento se utilizaron conjuntos de datos aleatorios con tamaños de $2^{\{10,11,\ldots,24\}}$, y costos de $c_{\text{máx}}$ de $\{10, 30, 50, 70, 90\}$, llevando a cabo la ejecución del algoritmo tres veces con dispositivos distintos en cada ocasión:

Componentes	Dispositivo 1	Dispositivo 2	Dispositivo 3
Sistema Operativo	Windows 11	Windows 11	macOS 13 (Ventura)
RAM	16 GB	8 GB	16 GB
Procesador	Intel(R) Core(TM) i5-1240P	No especificado	No especificado
Caché L1	1.1 MB	320 kB	No especificado
Caché L2	9 MB	2 MB	No especificado
Caché L3	12 MB	8 MB	8 MB

A continuación se presentan los distintos gráficos resultantes de aplicar el experimento en el dispositivo 1:

3.0.1. Gráfico de Costo Promedio Real vs Cantidad de Datos, para distintos $c_{\text{máx}}$

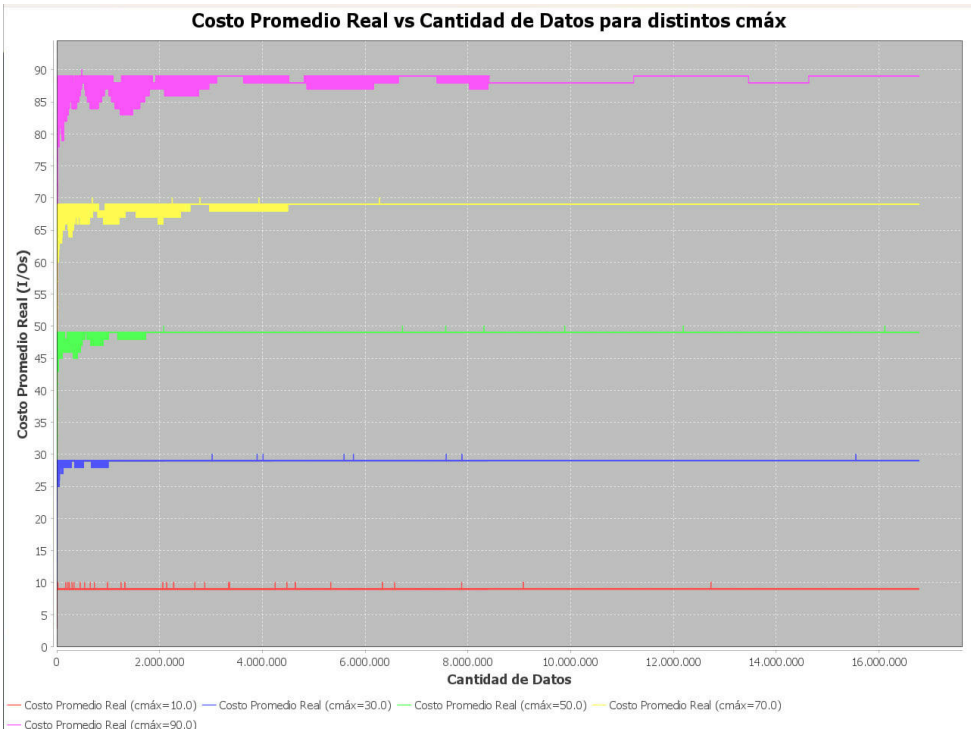

Figura 1: Valores de $c_{\text{máx}} \in \{10, 20, 30, 50, 90\}$.

3.0.2. Gráfico de Porcentajes de llenado vs Costo Promedio Real, para distintos $c_{\text{máx}}$

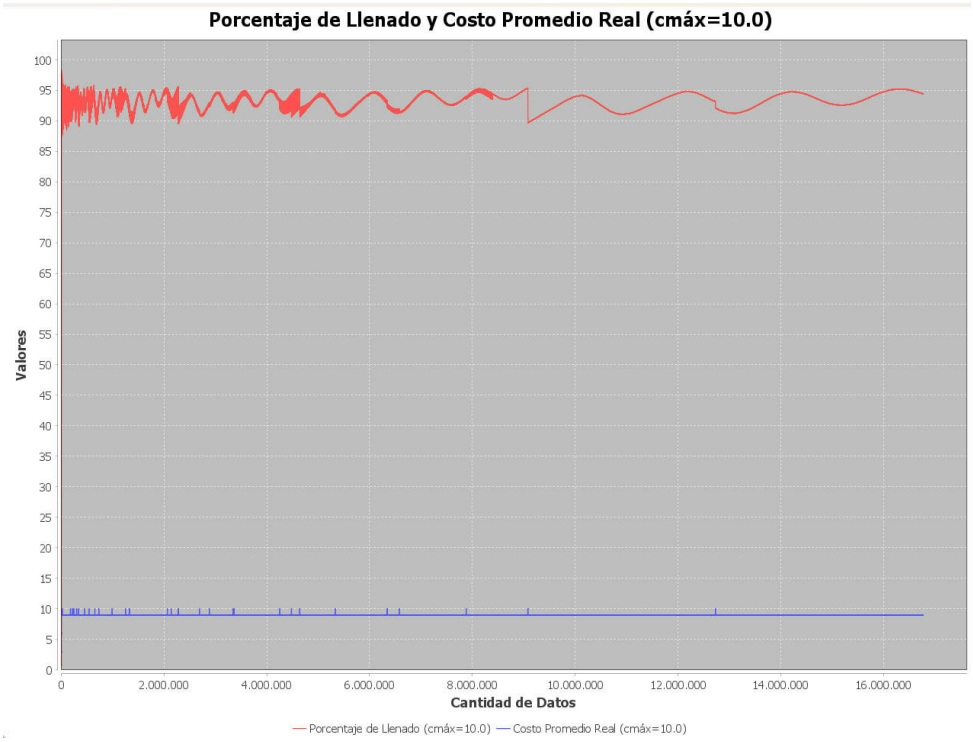

Figura 2: $c_{\text{máx}} = 10$.

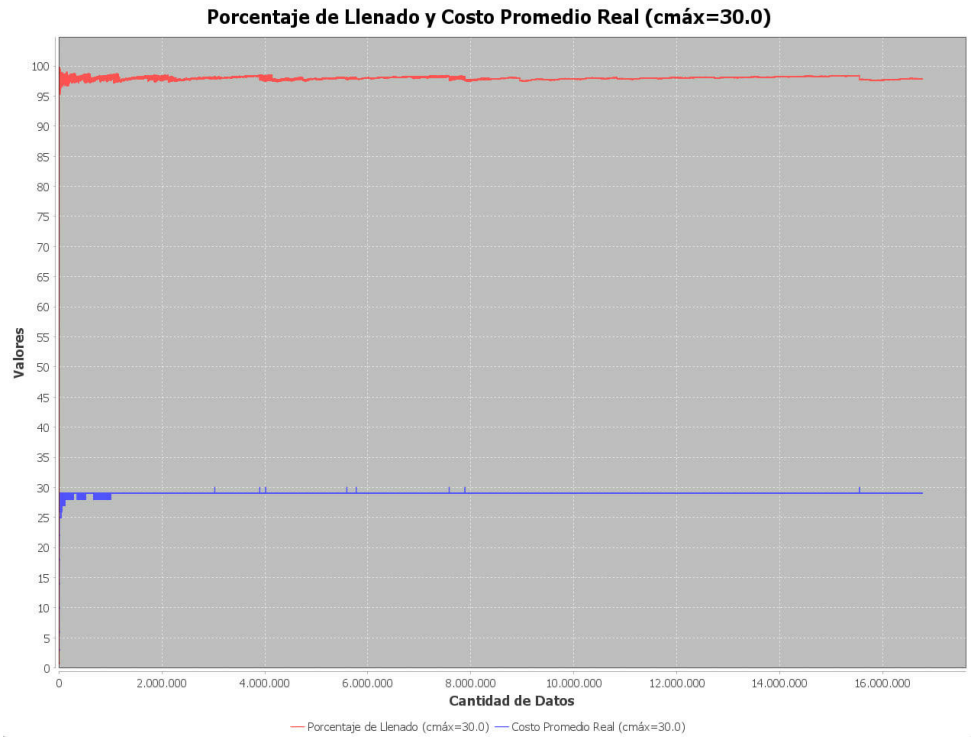

Figura 3: $c_{\text{máx}} = 30$.

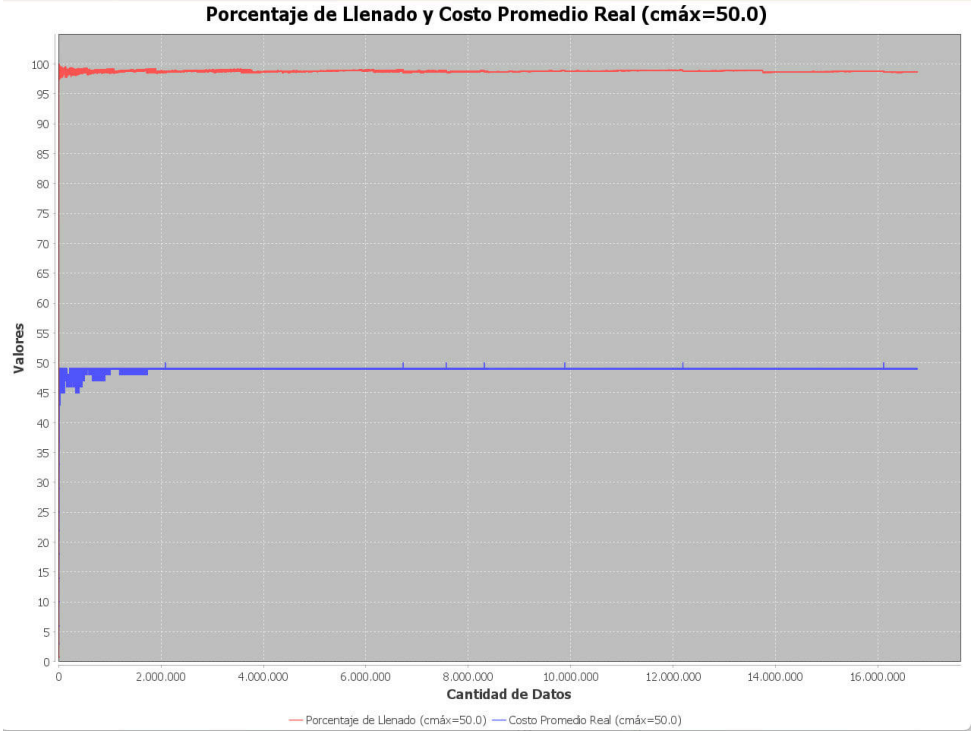

Figura 4: $c_{\text{máx}} = 50$.

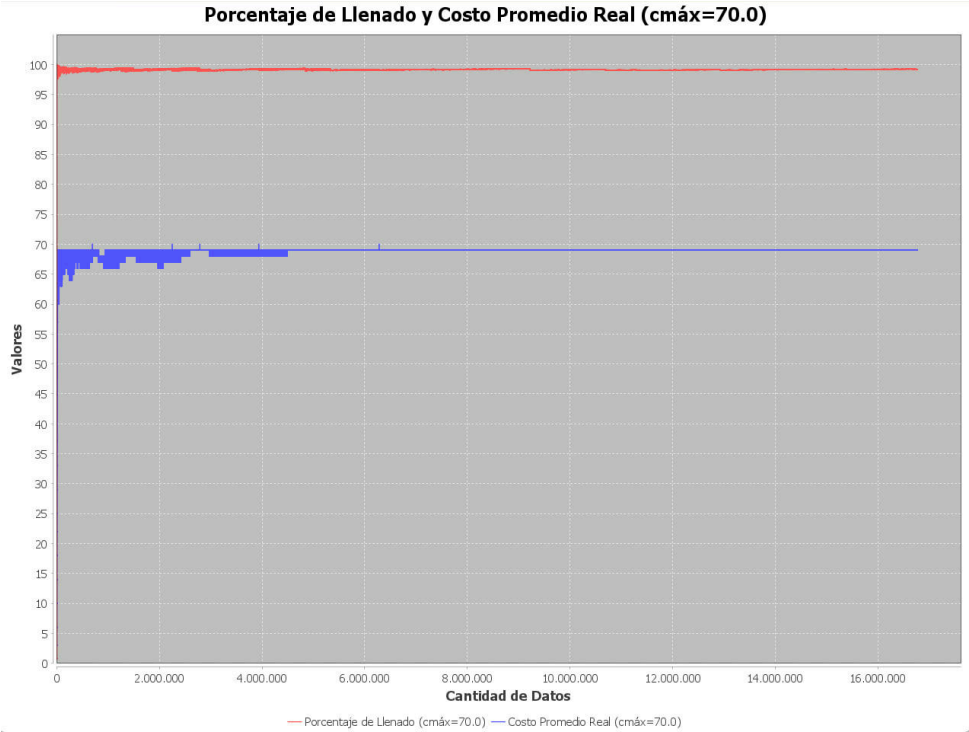

Figura 5: $c_{\text{máx}} = 70$.

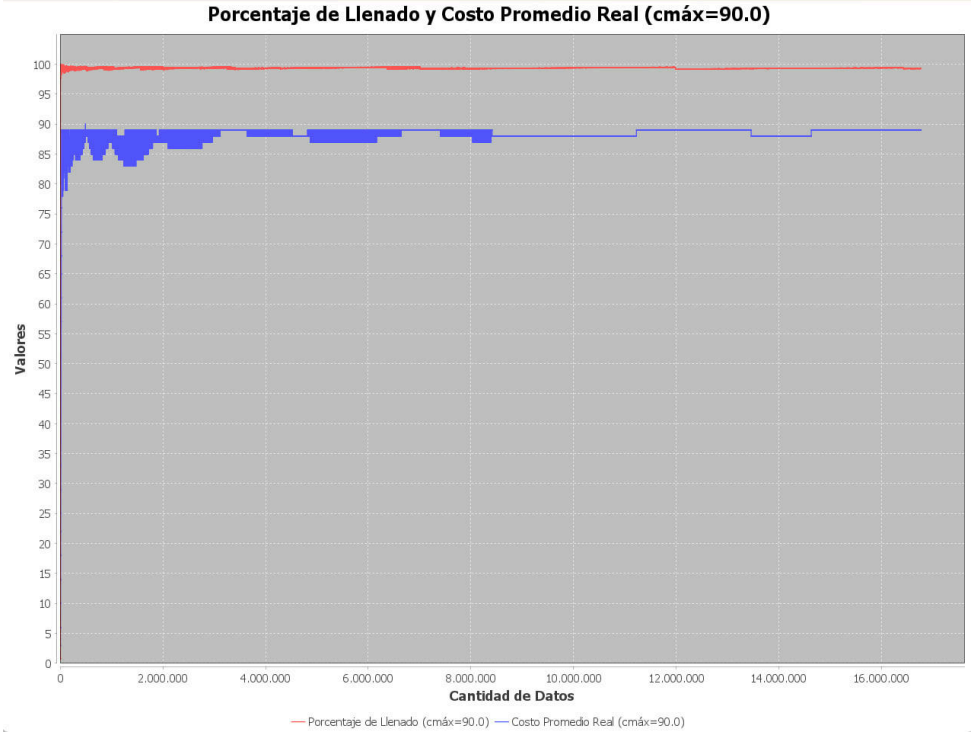

Figura 6: $c_{\text{máx}} = 90$.

4. Análisis

Para el costo promedio real respecto a la cantidad de datos, para distintos $c_{\text{máx}}$, es posible observar a partir de la figura 1 que a medida que $c_{\text{máx}}$ aumenta su valor, aumenta el rango de valores en donde el costo promedio real varía.

- $c_{\text{máx}} = 10$: El costo promedio real no supera los 12.5 I/Os para ninguna cantidad de datos, manteniéndose levemente bajo 10.
- $c_{\text{máx}} = 30$: El costo promedio real varía frecuentemente entre 25 y $\sim$30 para una cantidad de datos menor a 1.000.000, posterior a esta cifra el costo se estabiliza cercanamente a los 30.
- $c_{\text{máx}} = 50$: El costo promedio real varía entre 44 y $\sim$50 para una cantidad de datos menor a 2.000.000, posterior a esta cifra el costo se estabiliza cercanamente a los 50.
- $c_{\text{máx}} = 70$: El costo promedio real varía inicialmente entre 60 y $\sim$70, luego entre 65 y $\sim$70 para una cantidad de datos menor a 3.000.000, posterior a esta cifra el costo oscila entre 67 y $\sim$70 para datos menores a 4.500.000 y finalmente se estabiliza cercanamente a los 70.
- $c_{\text{máx}} = 90$: El costo promedio real varía inicialmente entre 78 y $\sim$89, estabilizándose alrededor de 85 para una cantidad de datos menor a 2.000.000. Luego el costo oscila frecuentemente entre 86 y 89, prácticamente estabilizándose entre 87 y 88, sin generar rebalses de páginas.

Otro aspecto a notar son los peaks para cada $c_{\text{máx}}$, a medida que esta variable controlada incrementa es posible notar un descenso de peaks, indicando que hay menor cantidad de ocurrencias en donde se debe expandir las páginas.

$c_{\text{máx}}$	**Peaks**
10	29
30	8
50	7
70	5
90	1

Por otra parte, a partir de los gráficos de las figuras 2 a la 6, que analizan la variación del porcentaje de llenado y la variación del costo promedio real para distintos $c_{\text{máx}}$ en función de los datos insertados, es observable que el comportamiento del costo promedio real es igual al explicado anteriormente. Respecto al porcentaje de llenado de páginas tenemos el siguiente análisis:

- $c_{\text{máx}} = 10$: Oscila entre 90 y 95, con una alta frecuencia al inicio.
- $c_{\text{máx}} = 30$: Oscila entre 97/98, con una frecuencia inicial menor a $c_{\text{máx}} = 10$ pero aún alta.
- $c_{\text{máx}} = 50$: Oscila entorno a 98, con una frecuencia más baja.
- $c_{\text{máx}} = 70$: Oscila en torno a 98/99, notándose una mayor estabilidad al tener menos frecuencia de una gran variación al inicio.
- $c_{\text{máx}} = 90$: Similar a $c_{\text{máx}} = 70$, oscilando en torno a 98/99 y con una menor frecuencia inicial con respecto a $c_{\text{máx}} = 10$.

Es decir, a medida que aumenta $c_{\text{máx}}$, el costo promedio real alcanza menos peaks, y el porcentaje de llenado se estabiliza cercanamente por debajo del 100% con menores variaciones.

Es notorio para $c_{\text{máx}} = 10$ y $c_{\text{máx}} = 30$ como los puntos de gran variación en el porcentaje de llenado coinciden con peaks de los costos promedio reales que alcanzan a la variable controlada.

5. Conclusión

A partir del análisis construido en base al experimento de Hashing Lineal, es posible realizar una serie de conclusiones:

- (1) Un $c_{\text{máx}}$ pequeño resulta más costoso para el algoritmo. Esto basado en que a medida que aumenta el $c_{\text{máx}}$ permitido, disminuye la cantidad de peaks, y por ende, de rebalses de páginas. Es decir, se realizan menos operaciones de rehashing.
- (2) Una vez que el costo promedio real alcanza la variable controlada del costo promedio máximo permitido, se realiza el proceso de expansión de páginas, lo que genera que porcentaje de llenado disminuya. Es decir, a medida que $c_{\text{máx}}$ aumenta, el porcentaje de llenado se estabiliza pues deja de haber momentos en lo que se liberan páginas mediante rehashing ya que el costo promedio real no gatilla esta acción.

Retomando nuestra hipótesis, esperábamos que este algoritmo no fuera tan eficiente debido a la condición de recorrer toda la lista al momento de insertar un elemento para verificar si esta ya se encontraba en ella. Esto fue demostrado al verificar que la gran cantidad de I/Os generaba que el costo promedio real alcanzara rápidamente a la variable controlada $c_{\text{máx}}$ para cantidades relativamente pequeñas de datos. Esto generaba una constante necesidad de vaciar páginas y aplicar el rehashing, observable en los gráficos de las figuras 2 a la 6, así como la relación de que estos peaks de costos reales gatillaban las variaciones en los porcentajes de llenado.

Pese a que el algoritmo no es del todo eficiente, cabe destacar que el comportamiento es notablemente mejor si se consideran solo $c_{\text{máx}} \geq 90$ y cantidades de datos superiores a los 9.000.000 .