Emanuel Estrada Larios
Prof. Jairo Salazar
Estructura de Datos
16 de noviembre del 2017

Ruleta 5: Linear Probe Hashing

Descripción de trabajo

Para el desarrollo de este programa, se utilizó el lenguaje Java para crear un programa que utilizara la estructura de universal hashing para ordenar artículos de la revista TecReview.

Para comenzar, se realizo una extensa investigación del método para entender mejor el funcionamiento y la implementación de la estructura y de nodos, utilizándose como referencia el libro de Cormen[1] y los artículos de S. Bargal[3] y Carnegie Melon University[4], los cuales describen la arquitectura de una tabla hash y como lidiar con colisiones con universal hashing en detalle.

Más adelante, se comenzó con el desarrollo del programa como tal, comenzando primero con el diseño del algoritmo y una vez terminado este, la implementación de una interfase gráfica. Se finalizó con una etapa de pruebas y debugging, así como con la realización de este reporte.

Evidencia de funcionamiento

Durante la etapa de pruebas, se probó el funcionamiento de la estructura por medio de un método main en la clase UniversalHashing, en la que se realizaron pruebas insertando, buscando y eliminando llaves manualmente. Todas las pruebas realizadas en esta etapa fueron exitosas.

Más adelante, se probó el funcionamiento con la interface gráfica y con los parámetros especificados por el profesor, de U=5 y S=3, y las pruebas fueron exitosos. En las pruebas de la clase se comprobó que el tamaño del hash puede ser mucho mayor y funcionar correctamente, pero se decidió utilizar este tamaño por dos razones: cumplir

con los parámetros especificados y simplificar el resultado del hash de las llaves para facilitar su interpretación.

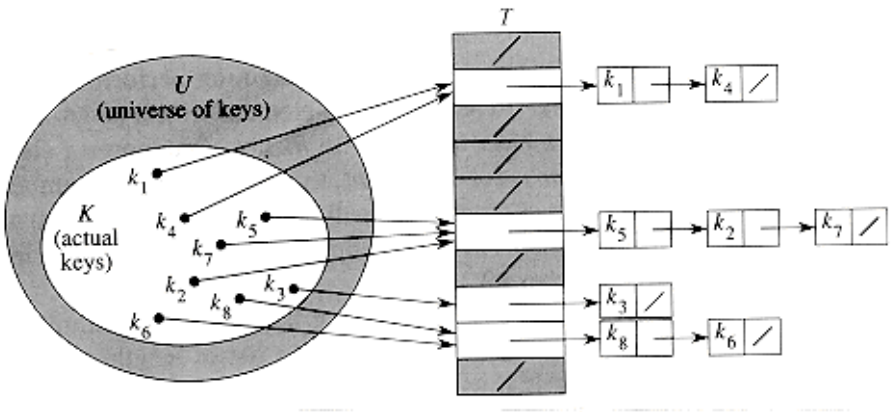
Si desea comprobar el funcionamiento del programa, puede obtener y compilar el código completo en GitHub[2].

Marco teórico

Las tablas hash son una estructura que ayuda a realizar operadores de diccionario de forma rápida y eficiente, porque utilizando una función se determina un índice único para guardar cada llave en un arreglo en la mayoría de los casos, y en caso de que se produzca un índice repetido, se puede resolver con diversos métodos.

Una tabla hash es en realidad un arreglo de llaves que se ordena mediante una función (comúnmente conocida como función hash), que utilizando la llave de cada elemento y una cierta operación, determina un índice para dicho elemento, y cuando un índice se produce más de una vez, lo cual se conoce como colisión, se puede resolver de varias maneras, en este caso se utiliza la técnica de universal hashing con listas ligadas.

El universal hashing es un método peculiar que busca reducir el número de colisiones a una probabilidad de menos de n/m (donde n es el número de elementos y m es el número de slots en la tabla) ya que en vez de solo tener una función hash, tiene una familia de funciones H, y cada vez se ingresa una llave, se elige una función aleatoria, reduciendo la posibilidad que llaves similares resulten en el mismo slot, y los elementos se distribuyen de forma más uniforme en la tabla. En el caso que se presente una colisión, se maneja con listas ligadas, agregando el nuevo elemento en el slot correspondiente y apuntando su atributo next al elemento anteriormente almacenado.



El diagrama anterior ilustra de manera sencilla el manejo de colisiones. Cuando más de una llave resulta en cierto índice, se almacenan en listas ligadas apuntadas desde el índice correspondiente. En esta implementación, el último elemento agregado será el primer nodo en la lista de izquierda a derecha, y su next será el elemento que se agregó previamente. Este algoritmo tiene una complejidad de $O(n)$ (dónde n = número de elementos en lista ligada en el índice) para búsqueda y eliminación, y $O(1)$ para insertar.

Código

```
1. class Key<T>{
2.   public int k;
3.   public T data;
4.
5.   public Key(int k, T data){
6.     this.k = k;
7.     this.data = data;
8.   }
9. }
```

Comentarios del código por líneas:

1-9: Clase genérica Key.
2-3: Declaración de los atributos k (identificador de llave) y data (dato satelital).
5-8: Constructor de la clase, donde se recibe como parámetro el identificador de llave y el dato y se guardan en sus respectivos atributos.

```
1. class LinearProbeHash {
2.
3.   private int h, u, m;
4.   private Random rHash;
5.   private Node[] hashTable;
6.
7.   private class Node{
8.     Node next;
9.     Key data;
10.
11.    public Node(Key key, Node next){
12.      this.data = key;
13.      this.next = next;
14.    }
15.
16.  }
```

```
17.  private int hash(Key key){
18.    int randomInt = rHash.nextInt(this.h);
19.    return (key.k * randomInt) % this.m;
20.  }
21.
22.  public int insertElement(Key key){
23.    int j = hash(key);
24.
25.    if(this.hashTable[j] == null){
26.      this.hashTable[j] = new Node(key, null);
27.      System.out.println("Insertado en: " + j);
28.      return j;
29.    }
30.    else{
31.      Node next = this.hashTable[j];
32.      this.hashTable[j] = new Node(key, next);
33.      System.out.println("Insertado en: " + j);
34.      return j;
35.    }
36.
37.  }
38.
39.  public int searchElement(Key key){
40.
41.    for(int j=0; j<this.hashTable.length; j++){
42.      if(this.hashTable[j] != null){
43.        if(this.hashTable[j].data.k == key.k){
44.          return j;
45.        }
46.        else{
47.          Node start = this.hashTable[j].next;
48.          while(start != null){
49.            if(start.data.k == key.k)
50.            return j;
51.            else
52.            start = start.next;
53.          }
54.        }
55.      }
56.    }
57.
58.    return -1;
59.  }
```

```
60.  public void deleteElement(Key key){
61.    int j = searchElement(key);
62.
63.    if(j != -1){
64.      if(this.hashTable[j].data.k == key.k){
65.        if(this.hashTable[j].next != null){
66.          this.hashTable[j] = this.hashTable[j].next;
67.        }
68.        else{
69.          this.hashTable[j] = null;
70.        }
71.      }
72.      else{
73.        Node primero = this.hashTable[j];
74.        Node segundo = this.hashTable[j].next;
75.        boolean borrado = false;
76.
77.        while(borrado == false){
78.          if(segundo.data.k == key.k && segundo.next == null){
79.            primero.next = null;
80.            borrado = true;
81.          }
82.          else if(segundo.data.k == key.k && segundo.next != null){
83.            primero.next = segundo.next;
84.            borrado = true;
85.          }
86.          else{
87.            primero = primero.next;
88.            segundo = segundo.next;
89.          }
90.        }
91.      }
92.    }
93.
94.  }
```

Comentarios del código por líneas:

1-94: Fragmento de la clase UniversalHashing.
3-5: Declaración de atributos, entre ellos el arreglo de nodos que contendrán las llaves.
7-16: Clase interna Node, que permite crear las listas ligadas, con atributos llave y next.
17-20: Función hash que elige un número aleatorio de la familia de hashes y regresa el índice donde se guardará la llave.

22-37: Método insertElement, que recibe como parámetro la llave a almacenar y regresa el índice donde se almacene el dato.

25-29: Condicional se detona cuando el índice de la tabla está vacío, y guarda el elemento en dicho índice.

30-35: En el caso contrario, se recorre el elemento actualmente almacenado un lugar en la lista y se inserta el nuevo elemento.

39-59: Método searchElement, que recibe como parámetro la llave a buscar y regresa su índice o -1 en caso de que el índice no se encuentre.

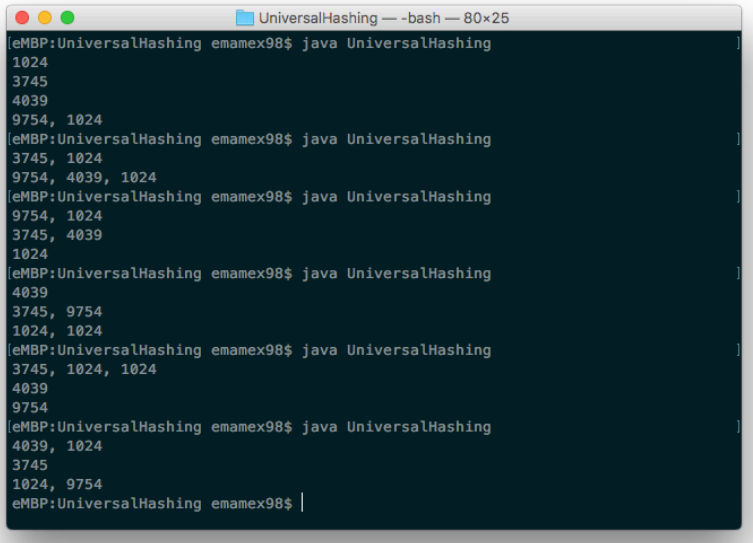
41-56: For loop recorre la tabla y las listas de cada índice. Esta implantación no es la más eficiente para tablas grandes, pero funciona bien con tablas pequeñas como con U=5.

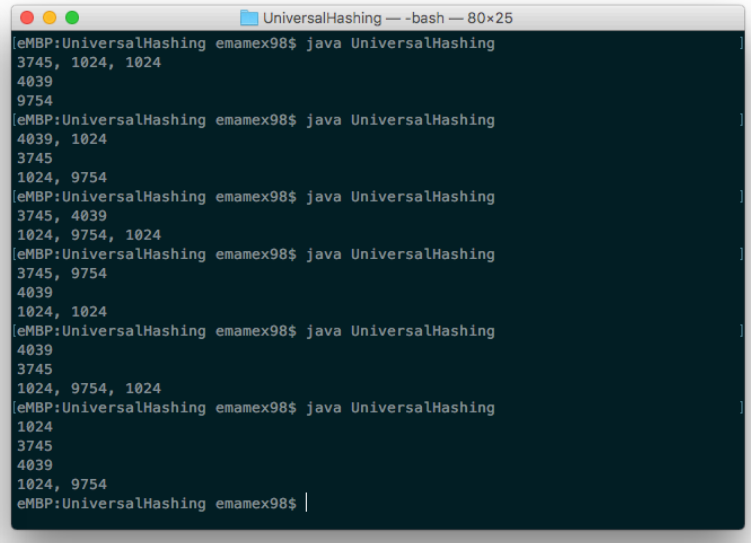
60-92: Método deleteElement, que recibe como parámetro una llave, la busca y si la encuentra, reajusta los apuntadores de los nodos, eliminando el elemento.

*Se omitieron algunos métodos que son irrelevantes a la funcionalidad principal de la estructura, como el método printTable, entre otros.

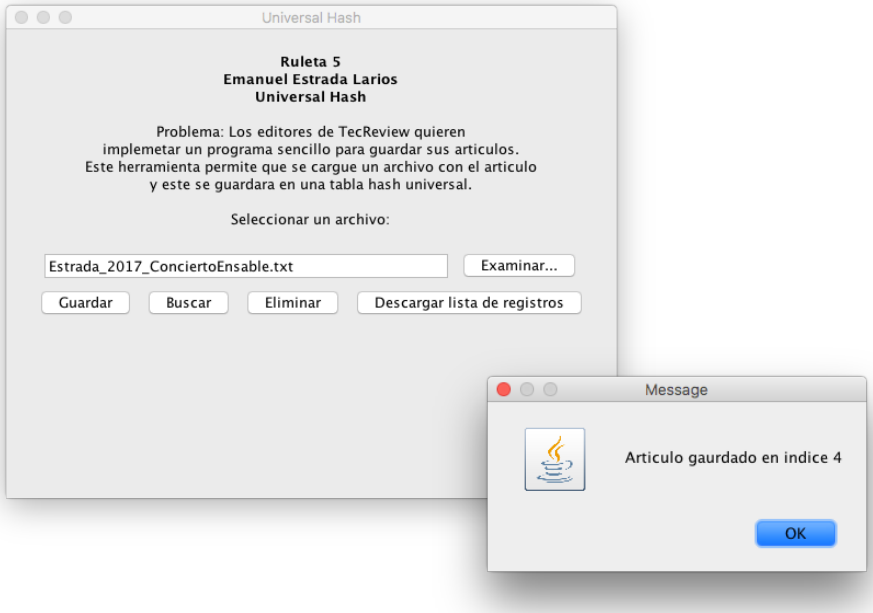
Casos de prueba

Como se mencionó anteriormente, se realizaron diversas pruebas para demostrar el funcionamiento del programa. En las primeras etapas del proyecto, se realizaron pruebas en la terminal en dónde se analizaba la distribución de las llaves en la tabla. Algunas de estas pruebas puedan observarse en las capturas de pantalla posteriores:

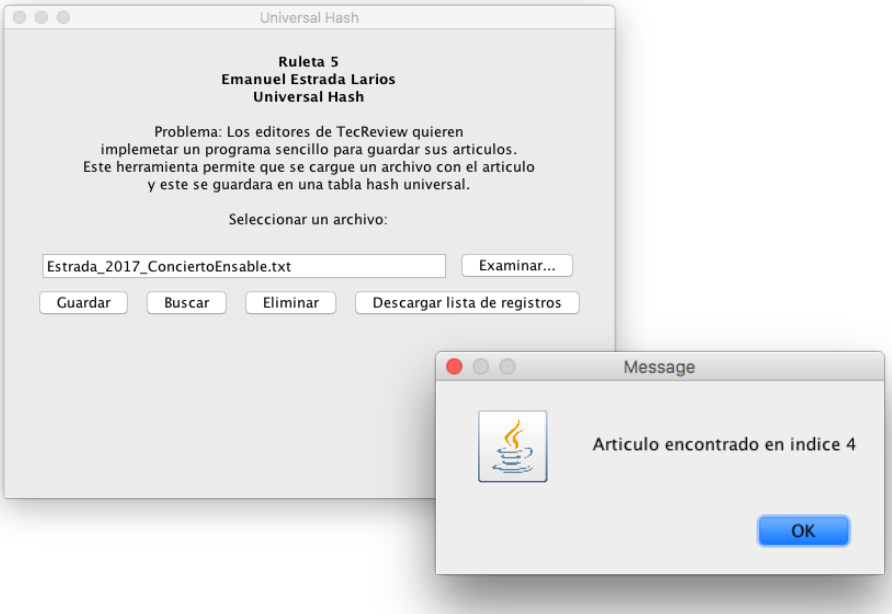




En las capturas de pantalla posteriores, se puede observar pruebas realizadas en la GUI. En la primera, se carga un archivo, y basado en su nombre se genera una llave que hashea al índice 4:



En la segunda captura de pantalla, se busca la misma imagen en el banco y es encontrada exitosamente.



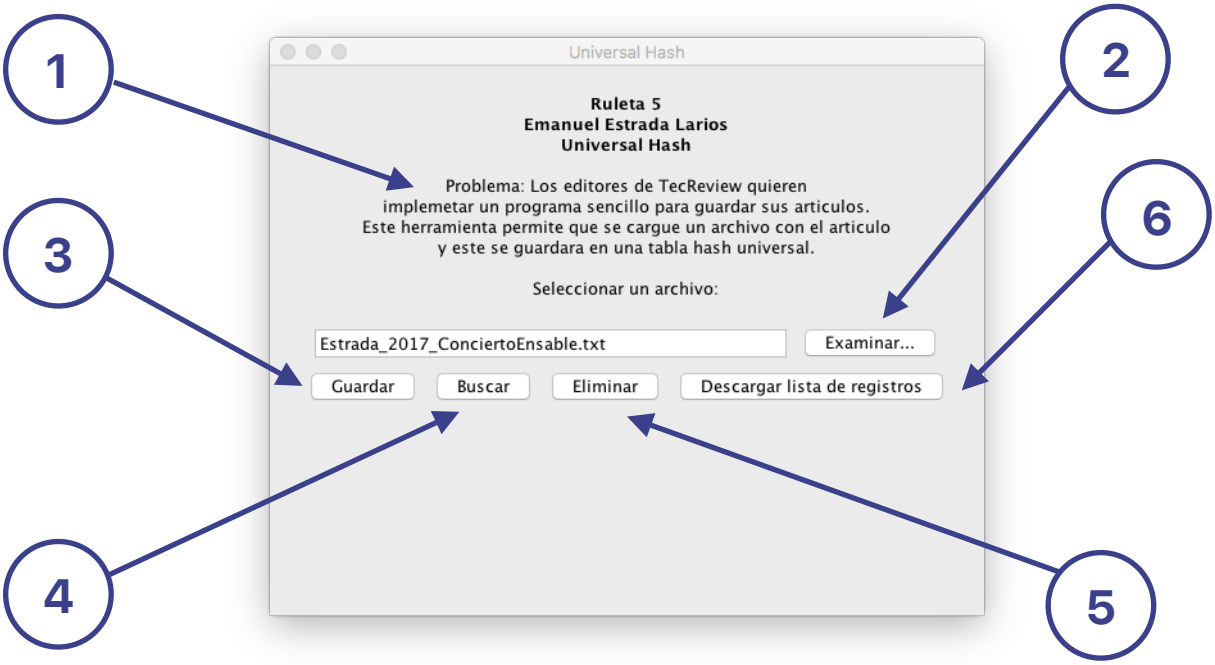
Si desea realizar sus propias pruebas utilizando el código que se encuentra disponible en GitHub[2] se proponen los siguientes casos:

1. Probar el programa utilizando la interface gráfica para probar las funciones "Agregar", "Buscar" y "Eliminar". Sin embargo, como ya se menciono antes, en la interface se limita el tamaño de la tabla conforme a los parámetros especificados por el profesor, pero esto puede ser modificado en el código fuente.
2. Durante las pruebas, no se excedió el límite de 100 de elementos en la lista, por lo que se recomienda trabajar con tamaños similares para garantizar funcionamiento exitoso.

Argumentos

Este programa resuelve el problema propuesto de manera óptima ya que el se reduce el número de colisiones por el hecho que se elige una función hash aleatoria. Además, el manejo de colisiones por medio de listas ligadas permite además reducir el tamaño de la tabla.

Descripción de la GUI



1. Presentación del problema, aplicación de la vida real y descripción del programa.
2. Botón para cargar un archivo, acompañado de un campo de texto en el que se muestra el nombre del archivo seleccionado.
3. Botón para agregar el archivo a la tabla hash.
4. Botón para buscar el archivo en la tabla hash.
5. Botón para eliminar el archivo de la tabla hash.
6. Botón para descargar la tabla en un archivo de texto.

Referencias y Recursos:

[1] "Hash Tables" in Introduction to Algorythms by Cormen, T.
[2] GitHub repository by Emanuel Estrada: https://github.com/emamex98/UniversalHash
[3] "Universal Hashing" by Sarah Adel Bargal: http://cs-www.bu.edu/faculty/homer/537/talks/SarahAdelBargal_UniversalHashingnotes.pdf
[4] "Universal and Perfect Hashing" by Carnegie Melon University: https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1004.pdf