Emanuel Estrada Larios
Prof. Jairo Salazar
Estructura de Datos
21 de septiembre del 2017

Ruleta 3: 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 linear probe hashing.

Para comenzar, se realizo una breve investigación del método para entender mejor el funcionamiento y la implementación de la estructura y de nodos, utilizándose como referencia los libros de Carmen y Sedgewick, los cuales describen la arquitectura de una tabla hash y como lidiar con colisiones con linear probing en detalle e incluye ejemplos de código.

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 del programa con diferentes tipos de datos –ya que la estructura y la clase key se construyeron de tipo genérico– incluyendo enteros positivos y negativos, strings y números con decimales, mas el valor de la llave siempre deben ser enteros. Se construyeron arreglos de diversos tamaños, la más grande de 1000 elementos, aunque es posible agregar más elementos. En todas estas pruebas, el resultado de las operaciones fue exitoso.

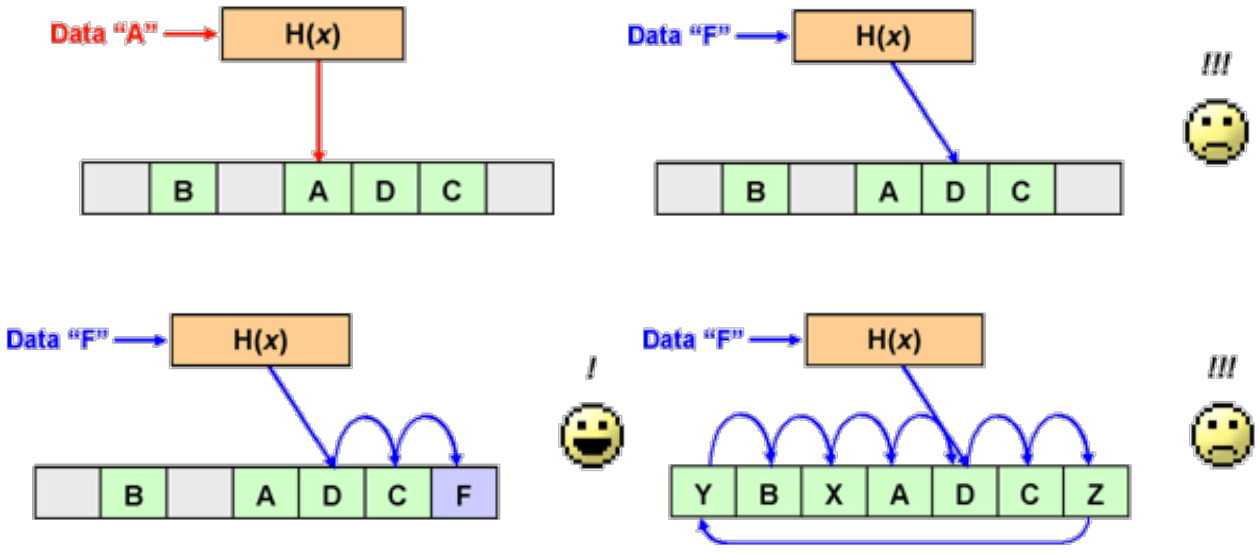
Sin embargo, es importante mencionar que aunque lo único que limita el tamaño de esta estructura es la memoria, en la interface gráfica de este programa se limita a un tamaño de 1000 elementos, lo cual puede ser modificado desde el código fuente.

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

Marco teórico

Las tablas de 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 una técnica de direccionamiento abierto, llamado linear probing. Esta técnica define su función hash de modo que el índice sea el resultado de la llave modulo tamaño del arreglo, y en caso de que el índice ya esté lleno, se va incrementando uno al valor de la llave, y se repite la operación cuantas veces sea necesaria. A esto se le llama número de probes, y por el hecho de que se le incrementa un valor de 1, es lineal.



Los diagramas anteriores ilustran el funcionamiento de la estructura. En la primera, se busca agregar el elemento A, por lo que se llama a hash que regresa un índice, y se agrega. En el segundo, se busca agregar el elemento B, pero al llamar hash, este regresa un índice que ya está ocupado por D, por lo que se le suma 1 y se recalca el hash hasta que regrese un índice desocupado. Si todos los índices ya están ocupados, entonces la función de agregar falla y regresa un error[3].

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 m, i, numberOfKeys;
4.   private Key[] hashTable;
5.   private Key deleted;
6.
7.   public LinearProbeHash(int tableSize){
8.     this.m = tableSize;
9.     this.hashTable = new Key[this.m];
10.    this.deleted = new Key(0,"");
11.    insertElement(deleted);
12.  }
13.
14.  private int hash(Key key){
15.    return (auxHash(key) + this.i) % this.m;
16.  }
```

```
17.  private int auxHash(Key key){
18.    return key.k;
19.  }
20.
21.  public int insertElement(Key key){
22.    this.i = 0;
23.    int j = 0;
24.
25.    while(this.i < this.m){
26.      j = hash(key);
27.      if(hashTable[j] == null || hashTable[j] == this.deleted){
28.        hashTable[j] = key;
29.        return j;
30.      }
31.      else{
32.        this.i++;
33.      }
34.    }
35.    return 0;
36.  }
37.
38.  public int searchElement(Key key){
39.    this.i = 0;
40.    int j = 0;
41.
42.    while(this.i < this.m){
43.      j = hash(key);
44.      if(hashTable[j] != null){
45.        if (hashTable[j].k == key.k)
46.          return j;
47.        else
48.          this.i++;
49.        }
50.      else
51.        break;
52.    }
53.    return 0;
54.  }
55.
56.  public void deleteElement(Key key){
57.    int j = searchElement(key);
58.    hashTable[j] = this.deleted;
59.  }
60.
61.}
```

Comentarios del código por líneas:

1-61: Fragmento de la clase LinearProbeHash.
3-5: Declaración de atributos, entre ellos el arreglo de llaves y la llave "deleted".
7-12: Constructor de la clase, donde también se agrega la llave "deleted" al índice cero que servirá como índice regresado al producirse un error.
14-19: Funciones hash y hash auxiliar, que calculan el índice del elemento utilizando el valor de su llave y los principios de linear probing.
21-36: Método insertElement, que recibe como parámetro la llave a almacenar y regresa el índice donde se almacene el dato.
25-34: While loop, que corre mientras el número de probes sea menor al tamaño de la tabla y mientras no se alcance un índice nulo o eliminado.
26-32: Se calcula el índice de la llave llamando a hash( ), y si este índice está disponible, se guarda la llave en el mismo, de lo contrario, se incrementa el número de probes para re-calcular el índice.
38-54: Método searchElement, que recibe como parámetro la llave a buscar y regresa el índice donde se almacene el dato.
42-52: While loop, que corre mientras el número de probes sea menor al tamaño de la tabla y mientras no se alcance un índice nulo o eliminado.
43-49: Se calcula el índice de la llave llamando a hash( ), y si el identificador de llave almacenado en este índice coincide con el del parámetro, entonces regresa el número de índice, de lo contrario, se incrementa el número de probes para re-calcular el índice.
56-59: Método deleteElement, que recibe como parámetro una llave, calcula su índice y le asigna la llave dummy "deleted".

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

```
1. private int calculateKey(String name){
2.     char[] charArr = name.toCharArray();
3.     int asciiKey = 0;
4.
5.     for(int i=0; i<charArr.length; i++)
6.       asciiKey += (int) charArr[i];
7.
8.     return asciiKey;
9.   }
```

Comentarios del código por líneas:

1-9: Fragmento de clase PeopleFinder; método calculateKey.
2-3: Conversión de string a char array y declaración de variable asciiKey.
5-6: Convierte cada carácter en valor numérico ASCii, y su sumatoria es almacena en asciiKey.

Casos de prueba

Como se mencionó anteriormente, se realizaron diversas pruebas para demostrar el funcionamiento del programa. En las capturas de pantalla posteriores, se puede observar pruebas realizadas en la GUI. En la primera, se agregó un nombre a la tabla, en la segunda se buscó el elemento, y en la última se eliminó el elemento.

También se llevaron a cabo otras pruebas con otro tipo de datos, como enteros positivos y negativos en la terminal.

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

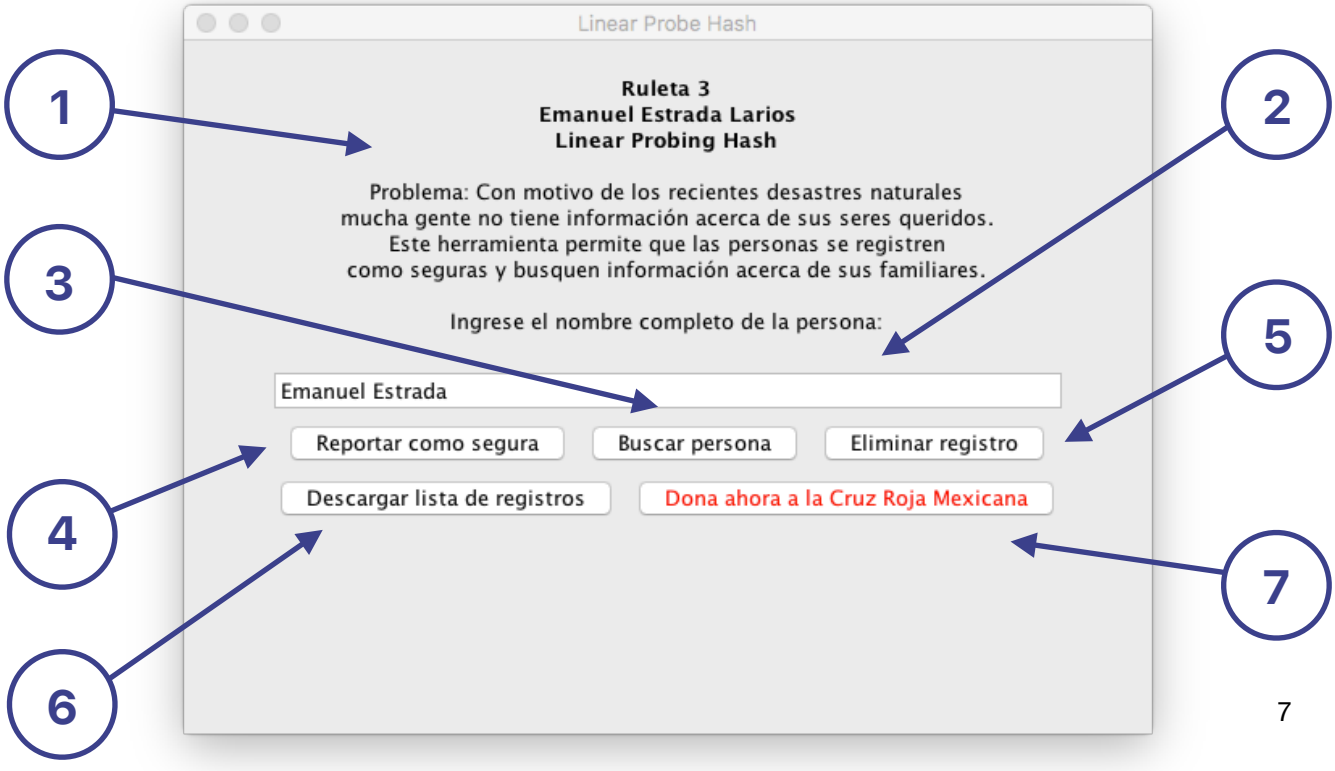
1. Probar el programa utilizando la interface gráfica para probar las funciones "Agregar", "Borrar" y "Buscar". Sin embargo, como ya se menciono antes, la interface limita el tamaño de la tabla a 1000 a menos que se modifique el código fuente.

2. Realizar pruebas con otros tipos de datos por medio de la creación de una clase de prueba con un método principal, evitando utilizar caracteres que no formen parte de la lista ASCii para generar una llave válida en el caso de utilizarse la función calculateKey.
3. Durante las pruebas, no se excedió el límite de 1000 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 tiempo de ejecución es mínimo ya que las operaciones no recorren por lo general el arreglo completo, sino que acceden directamente a los índices asignados por la función hash. La complejidad de tiempo de este algoritmo es $O(1r)$, donde $r = \alpha / (1 - \alpha)$, considerando que α representa el factor de carga, que se obtiene de $\alpha = n/m$ [4].

Descripción de la GUI



1. Presentación del problema, aplicación de la vida real y descripción del programa.
2. Campo de texto donde el usuario ingresa el nombre de la persona.
3. Botón que busca a la persona en la lista de registros.
4. Botón que agrega a la persona en la lista de registros de personas seguras.
5. Botón que elimina de la persona en la lista de registros.
6. Botón que genera un archivo de texto con los registros de la lista de personas.
7. Botón que invita al usuario a visitar el sitio web de donativos de la CRM.

Referencias y Recursos:

[1] "Hash Tables" in Introduction to Algorythms by Cormen, T.
[2] GitHub repository by Emanuel Estrada: https://github.com/emamex98/LinearProbeHash_ED
[3] "Linear Probing Hash Tables" by RMIT University: http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html
[4] "Hash table collision probability" by Johan: https://cs.stackexchange.com/questions/10273/hash-table-collision-probability