Tablas Hash

José Alberto Villa García
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv16c.jvilla@uartesdigitales.edu.mx

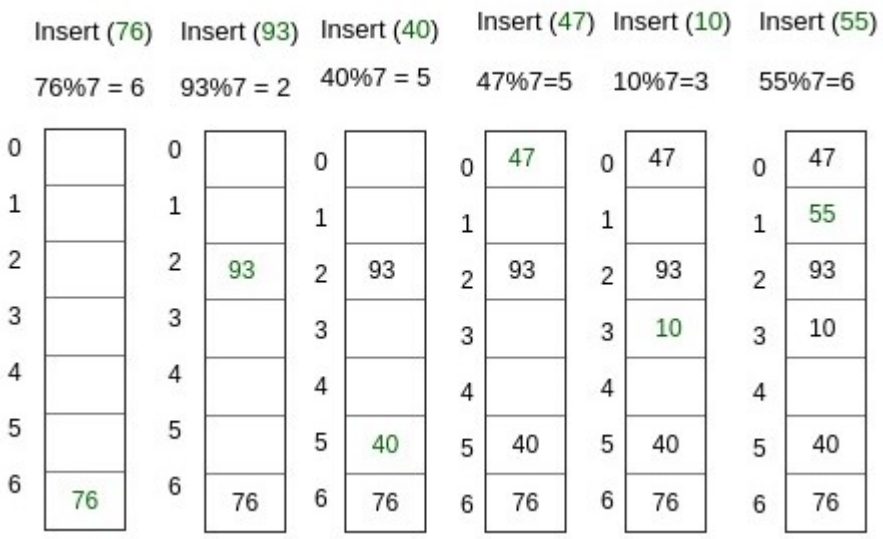
Profesor: Efraín Padilla

Julio 7, 2019

I. Introducción

Una tabla hash es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos almacenados a partir de una clave generada (usando el nombre o número de cuenta, por ejemplo). Funciona transformando la clave con una función hash en un hash, un número que identifica la posición (casilla o cubeta) donde la tabla hash localiza el valor deseado.

1) **Ejemplo:**



II. Código

A. *Nodo*

La clase nodo es una estructura que almacena una llave y un valor, los cuales son génericos, esto quiere decir que en teoría pueden ser del tipo que sea pero por el momento se asume que se usarán números (enteros o flotantes).
La estructura tiene un constructor con parámetros para poder llenarla desde que se crea.

Nodo.h C++:

```
template <typename K,
          typename V>
struct Node
{
 public:
  /*****************************************************************************************
  * Constructors
  *****************************************************************************************/

  /**
  *  @brief Default constructor
  */
  Node() = default;

  /**
  *  @brief Constructor with parameters
  *  @param key for this value
  *  @param actual value in the bucket
  */
  Node(K key, V value) : m_key(key),
                         m_value(value) {}

  /**
  *  @brief Default destructor
  */
  ~Node() = default;

  /*****************************************************************************************
  * Members
  *****************************************************************************************/

  /**
  *  @brief
  */
  K m_key;

  /**
  *  @brief
  */
  V m_value;
};
```

B. Table

Esta clase es la que contiene los nodos, también tiene dos tipos pero se asume que serán números. La clase tiene un constructor con parámetro de capacidad, lo cual determina cuántos valores puede almacenar la tabla y no en teoría no debería ser modificable.
Tiene los siguientes métodos:
* hashCode: Crea la key del valor a insertar
* insertNode: Inserta un valor a la tabla
* deleteNode: Borra un valor de la tabla
* get: Regresa el valor al que pertenece la key dada
* size: Regresa el tamaño de la tabla
* isEmpty: Regresa si la tabla tiene nodos o no

Table.h C++:

```
1
2   template <typename K,
3             typename V>
4   class Table
5   {
6    public:
7     /*****************************************************************************************
8     * Constructors
9     *****************************************************************************************/
10
11    /**
12    *  @brief Default constructor
13    */
14    Table();
15
16    /**
17    *  @brief
18    *  @param
19    */
20    Table(unsigned int capacity);
21
22    /**
23    *  @brief Default destructor
24    */
25    ~Table() = default;
26
27    /*****************************************************************************************
28    * Methods
29    *****************************************************************************************/
30
31    /**
32    *  @brief
33    *  @param
34    *  @return
35    */
36    int
37    hashCode(K key);
38
39    /**
40    *  @brief
41    *  @param
42    *  @param
43    */
44    void
45    insertNode(K key, V value);
46
47    /**
48    *  @brief
49    *  @param
50    *  @return
51    */
52    V
53    deleteNode(int key);
54
55    /**
56    *  @brief
```

```
    *  @param
    *  @return
    */
    V
    get(int key);

    /**
    *  @brief
    *  @return
    */
    int
    size();

    /**
    *  @brief
    *  @return
    */
    bool
    isEmpty();

    /**
    *  @brief
    */
    void
    display();

    /*****************************************************************************************
    * Members
    *****************************************************************************************/

    /**
    *  @brief
    */
    Vector<Node<K, V>*> m_array;

    /**
    *  @brief
    */
    int m_capacity;

    /**
    *  @brief
    */
    int m_size;
};

template <typename K,
          typename V>
void
HashTable::Table<K, V>::display() {
  //for(int i = 0; i < capacity; i++)
  //{
  //  if(m_array[i] != nullptr && m_array[i]->key != -1)
  //    cout << "key = " << m_array[i]->key
  //    << "  value = " << m_array[i]->value << endl;
  //}
}

template <typename K,
          typename V>
bool
HashTable::Table<K, V>::isEmpty() {
  return m_size == 0;
}

template <typename K,
          typename V>
int
HashTable::Table<K, V>::size() {
  return m_size;
}

template <typename K,
          typename V>
V
HashTable::Table<K, V>::get(int key) {
  int hashIndex = hashCode(key);
```

```
134   int counter = 0;
135
136   while(m_array[hashIndex] != nullptr)
137   {
138     int counter = 0;
139     if(++counter > m_capacity)
140       return static_cast<V>(-1);
141
142     if(m_array[hashIndex]->m_key == key)
143       return m_array[hashIndex]->m_value;
144     ++hashIndex;
145     hashIndex %= m_capacity;
146   }
147
148   return static_cast<V>(-1);
149 }
150
151 template <typename K,
152           typename V>
153 V
154 HashTable::Table<K, V>::deleteNode(int key) {
155   int hashIndex = hashCode(key);
156
157   while(m_array[hashIndex] != nullptr)
158   {
159     if(m_array[hashIndex]->m_key == key)
160     {
161       Node<K, V> *temp = m_array[hashIndex];
162
163       m_array[hashIndex] = nullptr;
164
165       --m_size;
166       return temp->m_value;
167     }
168     ++hashIndex;
169     hashIndex %= m_capacity;
170   }
171
172   return static_cast<V>(-1);
173 }
174
175 template <typename K,
176           typename V>
177 void
178 HashTable::Table<K, V>::insertNode(K key, V value) {
179   auto* temp = new Node<K, V>(key, value);
180
181   int hashIndex = hashCode(key);
182
183   while(m_array[hashIndex] != nullptr && m_array[hashIndex]->m_key != key
184         && m_array[hashIndex]->m_key != -1)
185   {
186     ++hashIndex;
187     hashIndex %= m_capacity;
188   }
189
190   if(m_array[hashIndex] == nullptr || m_array[hashIndex]->m_key == -1)
191   {
192     ++m_size;
193   }
194   m_array[hashIndex] = temp;
195 }
196
197 template <typename K,
198           typename V>
199 int
200 HashTable::Table<K, V>::hashCode(K key) {
201   return key % m_capacity;
202 }
203
204 template <typename K,
205           typename V>
206 HashTable::Table<K, V>::Table(unsigned int capacity) : m_capacity(capacity),
207                                     m_size(0) {
208   m_array.resize(m_capacity);
209 }
210
```

```
211   template <typename K,
212             typename V>
213   HashTable::Table<K, V>::Table() : m_capacity(20),
214                                     m_size(0) {
215     m_array.resize(m_capacity);
216
217   }
218 }
```