Árboles Binarios

Accel Israel Magaña Rodrgiuez

Universidad de Artes Digitales

Guadalajara, Jalisco

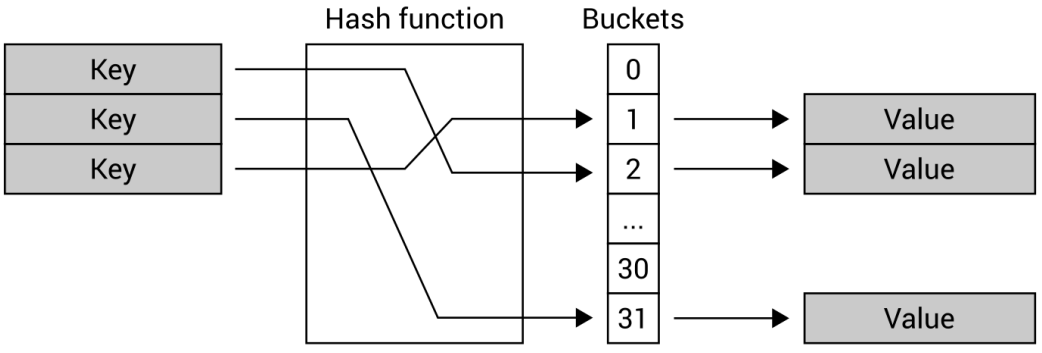
Email: idv17a.amagana@uartesdigitales.edu.mx

Profesor: Efraín Padilla

Julio 11, 2019

I. Hash Tables

El término hashing se refiere en realidad a un tipo de función con un fin dado (que vamos a ver más adelante) que se llaman funciones hash. Ésta es una función que dado un algoritmo mapea, un set de datos de longitud variable a otro set fijo.



Utilidad de las tablas

En seguridad:en criptografía, pero el más simple ejemplo de dónde se utilizan es para guardar contraseñas y buscarlas cuando el usuario necesite entrar.

En el transporte de información: se utiliza para chequeo de información y que coincida con uno original, si la función hash es distinta, entonces el archivo no es igual.

Estructuras de datos: esta aplicación es en la que estaré enfocado el parcial que viene, la de estructurar datos, específicamente en forma de diccionario y por pares.

Función Hash

La función hash se puede determinar como la función F(n) donde n es la key o lo que se requiere hacer hash, y la función internamente determina un número que será su posición en la tabla y cómo se va a guardar.

Existen varios tipos de funciones hash, ahora veremos 3 de estos:

- **Universal:** se refiere a "la mejor implementación" donde se quiere disminuir considerablemente el número de colisiones y además si introducimos un **K** varias veces, se espere la misma salida.
- **Multiplicativo:** consiste en operaciones dada la constante **C** que se define dentro de la misma función y es única. Sin embargo, esto puede causar más colisiones que el universal
- **Divisible:** el método divisible asume que los números enviados o la a **K** pueden ser divisibles y después se remueve el restante decimal para poder hacer un int hash.

Implementación en código C++:

```
template <typename K,typename V>
class HashMap
{
public:

  /**
   * @brief Actual constructor
   * @param size of map. DEFAULT = 100
   */
  HashMap(size_t size = 100): m_map(size), m_currSize(0){}

  /**
   * @brief Destructor
   */
  ~HashMap() {}

  /**
   * @brief Insert function
   * @param Key, how you gonna recognize the value send
   * @param Value, which value is going to be saved
   */
  bool insert(const K& key, const V& value);

  /**
   * @brief removes a value and key
   * @param key of which value gonna delete
   */
  bool remove(const K& key);

  /**
   * @brief returns the value
   * @param key of which value gonna return
   */
  V& operator[](const K& key);

private:
```

```
/**
 * @brief function where hash occurs
 * @param key that will be transformed into a int
 */
int myhash(const K& key) const {
  int hashed = 0;

  int street = 0;
  int house = 0;
  int person = 0;
  int digits = 0;

  float precision = 0.0f;

  street = key % m_map.size();
  house = std::abs(std::floor(key / m_map.size()));

  precision = static_cast<float>(key) / static_cast<float>(m_map.size());
  precision = std::fabs(precision) - house;

  person = static_cast<int>(precision * 1000);

  digits = street > 0 ? static_cast<int>(log10(static_cast<double>(street))) + 1 : 1;

  std::string toConcat;
  toConcat += std::to_string(digits);
  toConcat += std::to_string(street);
  toConcat += std::to_string(house);

  if(person < 10) { toConcat += std::to_string(0); }
  toConcat += std::to_string(person);

  toConcat += key > 0 ? std::to_string(1) : std::to_string(2);

  hashed = atoi(toConcat.c_str());
  return hashed;
}

/**
 * @brief the actual container where it'll be saved
 */
std::vector<std::pair<K, V>> m_map;

/**
 * @brief Actual size of table
 */
unsigned int m_currSize;
```

```
  typedef typename std::pair<K, V> mapIterator;

};


template <typename K, typename V>
V& HashMap<K, V>::operator[](const K& key)
{
  mapIterator& it = m_map[myhash(key)];

  if (it.first == key)
  {
    return it.second;
  }
}

template <typename K, typename V>
bool HashMap<K, V>::insert(const K& key, const V& value)
{
  mapIterator& it = m_map[myhash(key)];

  if(it.first != "")
  {
    it.first = key;
    it.second = value;
    return true;
  }

  //There's already a key like this one
  return false;
}
```