Hash Tables: The why and how.

Alessio Belaj
Data Structures
Professor Aizenman

May 20th, 2019

Introduction

So imagine that you wake up tomorrow and are assigned a mandatory position as a librarian. You must organize a library shelf with various books. Your job is to make these books easy to access by anyone willing to check them out. How will you do this? Well, you'd start by thinking of how someone would normally look for a book. Would they walk down the isles randomly and aimlessly looking for the book? No, that's inefficient and a waste of time. Ideally, you would want someone to be able to enter the book's title into a library computer and have the computer tell them EXACTLY where the book is. Which shelf number, which row, which column, whatever.

Well, we are actually able to do this efficiently using something called a hash algorithm. Using a hash algorithm, we are able to take in data(in this case, a book title) and output the book as a hash value. This hash value is unique to the book and if someone were to input the book title into the algorithm, it would give them the same hash value. This allows for easy access to that book. Essentially, this is how a hash table works.

Advantages and disadvantages of hash tables

The main advantage of hash tables over other data structures is speed and synchronization. They are the more efficient than search trees or any other table lookup structure. The disadvantage to hash tables is that hash collisions may occur. This happens

when the hash function produces the same result for two or more different pieces of data. This is why chaining occurs.

My project

Now that you have a better understanding of hash tables, I will explain what my project is on. Basically, I decided to use a hash table to implement an easily accessible dictionary. The program takes in a text file, which it reads line by line using ifstream, separates the key and the definition by a colon, and includes the key and definition into the hash table.

```
//read file for testing
string line;
ifstream test_file("dictionary.txt");
if (test_file.is_open())
{
  while (getline(test_file, line))
  {
    istringstream f(line);
    string s;
    vector<string> strings;
    while (getline(f, s, ':')) {
      strings.push_back(trim(s)); // token string and save
    }
    if (strings.size() == 2){
      // built dictionary
      hash_table.put(strings[0], strings[1]);
      words.push_back(strings[0]); // save word list
    }
  }
  test_file.close();
}
else {
  cout << "Unable to open test file";
  return -1;
}
```

More in-depth look at the code:

The zip file includes a main file (Dictionary.cpp), hashtable.cpp, a header file for the hashtable, and a text file of the definitions. What I did was create a header file that includes the declaration of a hash data class, and a declaration of a the actual hash table class. Said hash table class is comprised of a list of hash data objects that we named dictionary. The class contains various functions such as:

- size_t getHash(string value);
- void put(string key, string content);
- void remove(string key);

```
class HashTable
{
  list<HashData> dictionary;

  public:
    size_t getHash(string value); //get hash value
    void put(string key, string content); // inserts a key into hash table
    void remove(string key); // deletes a key from hash table
    void replace(string key, string definition); //replace definition in hash table
    bool checkKeyExist(string key);
    void printDict();
    string get(string key); // get definition
};
```

- void replace(string key, string definition);
- bool checkKeyExist(string key);
- String get(string key);

It is important to take into account what each function does.

size_t getHash(string value);

This function takes in a string “value” and turns it into an integer value, which becomes a hash value. It uses a default c++ function called “hasher”. The sole purpose of this function is to compute hash values.

```
size_t HashTable::getHash(string value){
  hash<std::string> hasher;
  return hasher(value); //computes hash values
}
```

Time complexity: O(1)

void put(string key, string content);

The put function doesn’t actually return anything. What it does is takes in a string “key”, and a string “content”, and first checks if the said key is existent. If so, it replaces the value in the hashtable. So, the purpose of the put function is to add a key to the hashtable.

Time complexity: O(1)

```
void HashTable::put(string key, string definition)
{
  if (checkKeyExist(key)){
    //if same key is exist, replace value in hashtable
    replace(key, definition);
  }
  else{
    size_t hash_key = getHash(key); //computes new hash key
    //insert data to dictionary
    HashData data;
    data.key = hash_key;
    data.definition = definition;
    dictionary.push_back(data);
  }
}
```

void remove(string key);

Similarly, this function doesn’t return anything either. It takes in a string

```
void HashTable::remove(string key)
{
  if (checkKeyExist(key)){
    HashData data;
    size_t hash_key = getHash(key);
    for (auto it = this->dictionary.begin(); it != this->dictionary.end(); it++) {
      if (it->key == hash_key) {
        data = *it;
        break;
      }
    }
    dictionary.remove(data);
  }
}
```

"key", checks if the key exists in the hashtable. If so, it assigns "getHash(key)" to hash_key. Then uses a for loop to iterate through the beginning and end of the dictionary. Inside the for loop, if the hash value is the same, then it assigns data to the pointer from the dictionary and removes the data. Long story short, the function removes a key from the hash table.

Time complexity: O(1)

void replace(string key, string definition);

Yet again, we have a function that doesn't return anything. Instead what it does, is take in a string "key", and a string "definition". It assigns "getHash(key)" to hash_key. From there, it loops through the dictionary, if the hash value is the same, then replace the old definition with the new one.

```
void HashTable::replace(string key, string definition){
  size_t hash_key = getHash(key);
  for (auto it = this->dictionary.begin(); it != this->dictionary.end(); it++) {
    if (it->key == hash_key) {
      it->definition = definition;
      return;
    }
  }
}
```

Time complexity: O(1)

bool checkKeyExist(string key);

This time, we have a boolean function which takes in a string "key", and returns true or false. What it first is to check if the dictionary is empty, to avoid any unnecessary work. If so, it will automatically return false. If not, it will continue the function and assign "getHash(key)" to hash_key. From there, it loops through the dictionary from begining to end. If the hash key is the same as "string key", then return true. In layman's terms, this function checks if a key already exists or not.

```
bool HashTable::checkKeyExist(string key){
  if (this->dictionary.empty()) return false;
  size_t hash_key = getHash(key);
  for (auto it = this->dictionary.begin(); it != this->dictionary.end(); it++) {
    if (it->key == hash_key) {
      return true;
    }
  }
  return false;
}
```

Time complexity: O(1)

string get(string key);

This function takes in a string “key”. It first checks to see if that key exists using the preexisting “checkKeyExist” function. If so, the get function returns the definition.

Time complexity: O(1)

```
string HashTable::get(string key){
  if (checkKeyExist(key)){
    size_t hash_key = getHash(key);
    for (auto it = this->dictionary.begin(); it != this->dictionary.end(); it++) {
      if (it->key == hash_key) {
        return it->definition;
      }
    }
  }
}
```

See attached source code for rest of the code