Hashing and Hashtables

Table of contents

Introduction

A hash table is a data structure used for a *Dictionary/Map* ADT that provides constant time insertion (even in the worst case) and near-constant time search (in the average case).

To understand how hashing and hashtable works, let's review the *key-indexed search* method, which gives us a constant-time search for **integer-only** keys. The setup requires an intermediate or an auxiliary array of length equal to the maximum key value (the space cost). To set up the auxiliary array we do the following: for each key in the input sequence, use the key as the index into the auxiliary array, and increment the value in that slot (or just set it to be non-zero if you're not concerned about sorting). To search for a key, we use the key as the index into the auxiliary array, and see if the value is non-zero (exists) or zero (does not exist).

The basic requirement of the key-indexed search is that the keys must be integers (and only integers, since the keys are used as indices into an array). The limitations are the following:

1. **`k` must not be very large compared to `n`**: If the range `k` is much larger than `n`, then the space cost may be too high. For example, if the input sequence is `‹5, 3, 7, 10^6›`, then we'll need an 10^6 element auxiliary array, which is obviously ridiculously too large for a 4-key input sequence.
2. **keys must be non-negative**: Since the keys must be valid indices, the keys must be ≥ 0. There is a trivial solution to this problem of course: shift all keys by the most negative key such that the smallest key becomes 0 (Java arrays use 0-based indexing). *Solved*.
3. **only keys, no values associated with keys**: If there are keys that have associated values, we lose the value if there are multiple occurrences of the same key with different values (we only count the number of times a key occurs, not the values associated with each occurrence of the key). The solution is to use an array of "buckets", where each bucket contains the `(key,value)` pairs for a particular key. We can implement the buckets using arrays (the auxiliary array is then an array of arrays), or using linked lists or chains (the auxiliary array is then array of lists or chains). The second method is often preferable. *Solved*.

We still have two problems:

1. the basic requirement that the keys be integers only, and

2. the limitation that k >> n causes the auxiliary array to be too large for practical use.

Hashing and hashtables solve both of these problems quite well, but in the process we lose the ability to sort. Let's start with the limitation that k must be reasonably small first, and then we'll deal with the non-integer keys.

Back to Table of contents

Basic Principle

The basic idea behind hashing is to use a "hash" function to map a key into an index (which must be a valid index into the auxiliary array known as the hashtable). Since we want to minimize the length of the hashtable, we need to perform some form of compression to map a very large key to a small valid index, which is a key step in hashing.

Back to Table of contents

Compression

If we could compress the keys into a smaller indices, then we could use a smaller auxiliary array. The idea is trivially simple - use a circular array. The simplest compression scheme is to use the modulus (%) operator to compress the keys so that the resulting index is a valid index into the auxiliary array. The hash function takes a key, computes the index using the modulus operator. If `m` is the number of slots in the auxiliary array:

```
hash(key) = key % m
```

which guarantees that the hash value is between 0 and m-1. Take the following sequence for example: `‹5, 3, 7, 10^6›` , If we use m = 11 (a prime number. why? see Issues later on), we can build the following hash table:

```
  0    1    2   3   4   5   6   7   8   9   10
-------------------------------------------------
|  | 10^6 |   | 3 |   | 5 |   | 7 |   |   |   |
-------------------------------------------------
```

Note: `hash(10^6)` returns 1.

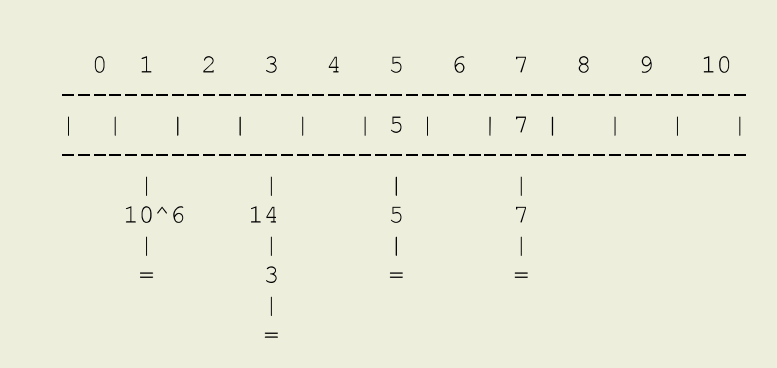
Now searching for a key can be done at constant time, and using very little space cost (for the auxiliary array).

Let's now add 14 to the sequence:

```
hash(14) = 14 % 11 = 3
```

ooops. There is already a key in that slot, and we have what's called a COLLISION. We've already seen something similar in key-indexed searching when we had non-distinct keys with (key,value) pairs, and

there we used array of buckets or chains to resolve the collision. We'll do the same here. Each array slot is now a linked list (probably using a dummy head reference). I'm only showing the non-empty chains below.



Why is 14 first in the list? (hint: where is it easy to add to a linked list?)

This particular implementation of a hashtable is called **separate chaining**. We've already seen this type of "arrays of lists" before, so there's really nothing new in terms of data structure here! We will look at other implementations later on in this document.

Back to Table of contents

Non-integer keys

Of course, we still have the basic requirement that the keys be integers. We now extend our hash function to map any arbitrary object to a large integer first, and then use compression to map it to a valid index. For example, if we have keys that are strings, our hash function would first somehow map each string to an integer, and then use compression to make it fit into the hashtable. See Chapter 14 of the textbook for details. Java makes it easy by providing a `hashCode()` method for each object (which returns a large integer, which we then need to compress to fit into our auxiliary array), so our hash function becomes:

```
public static int hash(Object o, int m) {
    int slot = o.hashCode() % m;
    return slot;
}
```

One thing to note that is that `hashCode()` may return a negative number (search google to see why), so we'll have to modify our code a bit:

```
public static int hash(Object o, int m) {
    int slot = o.hashCode() % m;
    if (slot < 0)
        slot = m + slot;
    return slot;
}
```

Back to Table of contents

Types of hashtables

Two common types of hashtables, each uses a different technique for resolving collisions:

1. Separate chaining: implementing the "buckets" using linked lists.
2. Open addressing: in case of collision, "probes" to see the next empty location where to put the key. Works well, but deleting a key is fairly complicated.

Back to Table of contents

Issues

1. How big should a hashtable be? Well, it should be bigger than "n". The idea is that the **load factor** n/m should be fairly small so that each chain is sufficiently short to make the search constant time. If the load factor becomes too high, you'll have to create a larger hash table and re-hash all the existing keys into this larger hash table.
2. Why a prime number? To avoid having patterns in the input sequence that produces collisions. For example, with m = 10, and the input sequence of < 0, 10, 20, 30, 40, 500, 1000, 23500 > produces 7 collisions with 8 keys!
3. Why do we need to rehash when resizing a hashtable? Because the compression uses the length of the hashtable, the indices into the older table won't match the indices into the new table.
4. There is no quick way to get the keys back in sorted order from a hashtable. Contrast this with a dictionary implemention using binary search tree, where a simple in-order traversal produces sorted keys as output. No choice but to extract the keys (using iteration), and then use an external sorting algorithm to get the keys in sorted order.

   Likewise, there is no quick way to tell what a successor or predecessor of a key in the table is (useful if you're looking for the keys closest to some key).

Back to Table of contents

Data Structures and implementation

For the separate chaining, we use the same data structure we've used for Adjacency List for Graphs. For open addressing/probing, we use a simple array.

Back to Table of contents

Discussion

Back to Table of contents