Hashing

Contents

Introduction

Recall that for a balanced tree (e.g., a red-black tree), the insert, lookup, and delete operations can be performed in time logarithmic in the number of values stored in the tree. Can we do better? Yes! We can use a technique called **hashing** that is logarithmic in the worst case, but has expected time O(1)!

The basic idea is to store values (unique keys plus perhaps some associated data) in an array, computing each key's position in the array as a function of its value. This takes advantage of the fact that we can do a subscripting operation on an array in constant time.

For example, suppose we want to store information about 50 employees, each of whom has a unique ID number. The ID numbers start with 100 and the highest ID number for any employee is 200 (i.e., there are a total of 101 possible ID numbers, all in the range 100 to 200). In this case, we can use an array of size 101 and we can store the information about the employee with ID number k in array[k-100]. The insert, lookup, and delete operations will all be constant time. The only disadvantage is that some of the array entries will be empty (i.e., will contain null to indicate that no information is stored there), so some space will be wasted.

Before we go on, here is some terminology:

- The array is called the **hashtable**.
- We will refer to the size of the array as **TABLE_SIZE**.
- The function that maps a key value to the array index in which that key (and its associated data) will be stored is called the **hash function**. For this example, the key is the employee's ID number, and the hash function is: hash(k) = k - 100.

Now, think about the problem of storing information about ***students*** based on their ID numbers. The problem is that student ID numbers are in a ***large*** range (student IDs have 10 digits, so there are 10^{10} possible values). In this case, it is probably not practical to use an array with one place for each possible value.

The solution is to use a "reasonable" sized array (more about this later) and to use a hash function that maps ID numbers to places in that array. If we can find a hash function that, given only a small set of ID numbers, is likely to map each ID number to a different place in the array, then we still have fast lookup, insert, and delete operations without requiring a huge array!

Here's an example: Suppose we decide to use an array of size 10 and we use the hash function:

hash(ID) = sum of digits in ID mod 10

Here are some ID numbers, the sums of their digits, and the array indexes to which they hash:

ID	Sum of digits	Array index (sum of digits mod 10)
9014638161	39	9
9103287648	48	8
4757414352	42	2
8377690440	48	8
9031397831	44	4

Note that we have a problem: both the second and the fourth ID have the ***same*** hash value (8). This is called a **collision**. How can we store both keys in array[8]? The answer is that we can make the array an array of linked lists, or an array of search trees, so that in case of collisions (if multiple keys have the same hash value), we can store multiple keys in the same place in the array. Assuming that we use linked lists, here's what the hashtable looks like after the 5 ID numbers given above have been inserted:

```
  [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
+---+---+---+---+---+---+---+---+---+---+
| \ | \ |   | \ |   | \ | \ | \ |   |   |
|  \|  \| | |  \| | |  \|  \|  \| | | | |
+---+---+-|-+---+-|-+---+---+---+-|-+-|-+
          |       |               |   |
          v       v               v   v
    4757414352  9031397831 8377690440 9014638161
                                  |
                                  |
                                  v
                           9103287648
```

TEST YOURSELF #1

Consider storing the names: George, Amy, Alan, and Sandy in a hashtable of size 10, using the hash function:

hash(name) = sum of characters mod 10

where a = A = 1, b = B =2, etc. Draw the hashtable that would be produced.

solution

Two important questions are:

- How should we choose the size of the hashtable (the array)?
- How should we choose the hash function?

The answers will be discussed below. First, let's assume that the hashtable size is TABLE_SIZE and that we have a hash function, and let's consider what the lookup, insert, and delete operations will do.

Lookup, Insert, and Delete

To look up a key k in a hashtable, all you have to do is compute k's hash value (v = hash(k)), then see if k is in array[v]. As mentioned above, the array will contain linked lists, or possibly search trees. In either case, you should already know how to look for k. The time for lookup will be proportional to the time for the hash function, plus the time to look for k in the data structure in array[v]. In the best case, when at most one key hashes to each location in the table, lookup in array[v] will be O(1). In the worst case, ***all*** of the keys will hash to the same place. In that case, if linked lists are used, the worst-case time for lookup will be O(N), where N is the number of values stored in the hashtable. If a balanced search tree is used, the time will be O(log N).

Inserting a key k in a hashtable is similar to looking it up: first, v = hash(k) is computed, then k is added to array[v]. If linked lists are used, k should be added at the front of the list (since that can be done in constant time). The time for insert is similar to the time for lookup: the sum of the time for the hash function and the time to insert k into array[v]. However, if linked lists are used, the time to insert k into the array will always be O(1) rather than O(N) in the worst case.

To delete a key k from a hashtable, v = hash(k) is computed, then k is deleted from the linked-list / search tree in array[v]. The worst-case time is the same as for lookup, since the value has to be found before it can be deleted.

Choosing the Hashtable Size

The best size to choose for the hashtable will depend on the expected number of values that will be stored and how important space consumption is (there will be a trade-off between the amount of space used and the number of keys that hash to the same array index). It is reasonable to use a table that is a bit larger than the expected number of items (say 1.25 times the expected number). If the number of items to be stored is not known, then you can always plan to expand the hashtable whenever it gets too full.

Choosing the Hash Function

The important issues to consider when choosing a hash function are:

- The computation of the hash function should be ***efficient*** (i.e., should not take too long).
- The hash function should spread the key values as evenly as possible (i.e., should map different keys to different locations in the hashtable).

Since the result of the hash function will be used as an array index, it must be in the range 0 to TABLE_SIZE-1. Therefore, it is reasonable for the hash function to convert the key to an integer n and to return n mod TABLE_SIZE.

If the keys ***are*** integers, with well distributed values (i.e., modding them with TABLE_SIZE is likely to produce results evenly distributed from 0 to TABLE_SIZE-1), then the hash function can just return key mod TABLE-SIZE. However, if the keys are not well distributed or if they are strings (or some other non-integer type), then we must convert them to well-distributed integer values (before modding them by TABLE_SIZE).

Let's assume that the keys are strings. Most programming languages provide a way to convert individual characters to integers. In Java, you can just cast a `char` to an `int`; for example, to convert the first character in `String S` to an `int`, use:

```
int x = (int)(S.charAt(0));
```

Once we know how to convert individual characters to integers, we can convert a whole string to an integer in a number of ways. First, though, let's think about ***which*** characters in the string we want to use. Remember that we want our hash function to be ***efficient*** and to map different keys to different locations in the hashtable. Unfortunately, there tends to be tension between those two goals; for example, a hash function that only uses the first and last characters in a key will be faster to compute than a hash function that uses all of the characters in the key, but it will also be more likely to map different keys to the same hashtable location.

A reasonable solution is to compromise; e.g., to use N characters of the key for some reasonable value of N ("reasonable" will depend on how long you expect your keys to be and how fast you want your hash function to be). For keys that have more than N characters, there is also a question of which characters to choose. If the keys are likely to differ only at one end or the other (e.g., they are strings like "value1", "value2", etc., that differ only in their last characters), then this decision could make a big difference in how well the hash function "spreads out" the keys in the hashtable.

To simplify our discussion, let's assume that we know the keys won't be too long, so our hash function can simply use all of the characters in the keys.

Here are some ways to combine the integer values for the individual characters to compute a single integer n (remember that the hash function will return n mod TABLE_SIZE):

- Add the integer values of all of the characters. This method is not very good if the keys are short and the table size is large because summing the integer values of the characters may not ever produce large indexes, so many spaces in the hashtable will never be used and other spaces will have to store many keys. Furthermore, it will hash all **permutations** to the same value. For example, hash("able") will be the same as hash("bale").

- Add the integer values of the characters, but multiply intermediate results by some constant to make the values larger. For example, suppose the key is "hello" and the integer values of the letters are: a = 1, b = 2, c = 3, etc. The values for the characters in "hello" are: 8, 5, 12, 12, 15. We can add the values, multiplying each intermediate result by 13:

  ```
  (8+5) * 13  = 169
  (169+12) * 13  = 2353
  (2353+12) * 13 = 30745
  (30745+15) * 13 = 399880
  ```

  This technique gives you a wider range of values than just adding all of the characters. However, you would have to be prepared to handle overflow. (You would probably want to test the value of the sum so far, and if it is too large, divide by some constant, making it smaller to prevent overflow.)

- Multiply individual characters by different constants and then add. For example, we could multiply the value of the first character by 8, the second character by 4, and the third character by 3. Then start again: multiply the fourth character by 8, the fifth by 4, and the sixth by 3, etc., and finally add up all of the products. This is less likely to lead to overflow than the previous technique and (unlike the first technique) will usually map permutations to different values.

TEST YOURSELF #2

Consider hashing keys that are strings containing only lower-case letters. The hash function that will be used computes the ***product*** of the integer values of the characters in a key, using the scheme: a = 0, b = 1, c = 2, etc. Why is this scheme not as good as using: a = 1, b = 2, etc.?

solution

Although it is a good idea to understand the issues involved in choosing a hash function, if you program in Java, you can simply use the `hashCode` method that is supplied for every `Object` (including `Strings`). The method returns an integer j, and you can just use j mod TABLE_SIZE as your hash function. Of course, if your keys are members of a class that you define, you will need to implement the `hashCode` method for that class yourself.

Summary

- Given a fixed table size and a hash function that distributes keys evenly in the range 0 to TABLE_SIZE-1, the expected times for insert, lookup, and delete are all O(1), as long as the number of keys in the table is less than TABLE_SIZE.
- Using balanced trees as array elements, the worst-case times for insert, lookup, and delete are all O(log N), where N is the number of keys stored in the table.
- Using linked lists as array elements, the worst-case times are O(1) for insert and O(N) for lookup and delete.
- A ***disadvantage*** of hashtables (compared to binary-search trees and red-black trees) is that it is not easy to implement a print method that prints all values in sorted order.

Back

Next: Continue with Sorting