Objectives

Direct-address tables
Hashing
Hash functions
Collision Resolution
    Chaining
    Linear Probing
    Quadratic Probing
    Double Hashing
Rehashing
Go over the Hashing project

Book Sections

Chapter 20

Introduction

BSTs provide fast insert, find, and delete operations that run in O(log N) time. The BST also keeps the values in sorted order. Is it possible to create a data structure that improves on the BST's O(log N) performance? The answer is "yes". We can use a technique called "hashing" to achieve insert, find, and delete operations that run in constant time on average. The tradeoff is that hash tables do not keep the values in sorted order, so hashing can only be used if we don't need sorting. Hashing also uses more memory than BSTs.

Direct-Address Tables

Before we talk about hashing, we first discuss a related technique called "Direct-Address Tables".

Consider the case where the values being stored in the collection are integers that are fairly limited in range. For example, suppose that we are storing employee ID numbers, and all employee numbers are integers less than 10,000. How can we store the employee numbers so that they can be searched in time faster than O(log n)?

We could simply create an array of 10,000 elements. To store a particular employee number, we simply use the employee number as an index into our array, and store the number at the corresponding array position. When searching for a particular employee number, we follow the same process by using the employee number to index into the array, and we check that position to see if the number is in the collection. Deletion from the collection is similar.

This approach is called "direct-address tables". It provides O(1) insert, lookup, and delete operations. In fact, the most expensive operation with this approach is the initialization of the array, which takes time proportional to the size of the array.

If the values being stored are not integers (e.g., they might be character strings or objects), we must find a way to convert each value into a unique integral value.

When doesn't the direct-address table approach work?

Direct-address tables are only practical when the number of possible values is relatively small. Otherwise, if the number of possible values is very large, it will be impractical to create an array that is large enough to store every possible value. This is especially true if the number of values actually being stored is much less than the number of possible values. In such cases, even if it is possible to create an array that is large enough to hold all possible values, it would be very wasteful of space to do so.

Hashing

Direct-address tables provide very good performance for the dictionary operations, but that they are not always practical when the value space is large. There is another approach called "hashing" that has performance characteristics that are very similar to those of direct-address tables, but that can be used no matter how large the value space.

Like direct-address tables, hashing also uses an array to store the values in the collection. We call this array the "hash table".

When the value space is very large, we cannot directly use the value as an index into the hash table. Instead, we pass the value to a function called a "hash function" which takes returns the index in the hash table that should be used to store the value. We call this the "hash value" for that particular value. Hash functions are designed to be fast so that we can efficiently decide which hash table entry corresponds to a particular value.

Once we have computed the hash value for the value, we can easily index into the hash table and perform the requested insert, lookup, or delete operation.

For example, with our employee database, suppose that employee numbers can have values up to 1 million. In this case, we might create a hash table with 10,000 elements, and use a hash function that returns the last four digits as the index in the hash table for that value.

The advantage of hashing over direct-address tables is that a hash table need not contain an entry for every possible value. This is what makes hashing practical when direct-address tables are not.

The disadvantage of hashing, however, is that multiple values can now map to the same entry in the hash table. In other words, the hash function can return the same hash value for multiple values, which we call a "collision". Since we are only able to store one value in each hash table entry, when a collision occurs we are forced to find a different place to store the value.

Of course, the ideal solution is to design a hash function so that there are no collisions. However, since the hash table has fewer entries than the number of possible values, it is not possible to avoid collisions entirely. Therefore, our hash table algorithms must be able to deal with collisions.

Even though we can't avoid collisions entirely, a good hash function will minimize the number of collisions by making them as unlikely as possible. A good hash function will distribute values evenly throughout the entire hash table in a seemingly random fashion. Even if two values are similar, they should hash to entirely different places in the hash table.

For example, suppose that in our employee database, each employee's ID number ends with four digits representing the year in which they were hired. In this case, using the last four digits as the hash value would be disastrous.

Hash Functions

Hash Functions for Integers

If the values are integers, then implementing the hash function is easy. Simply take the value and mod it by the size of the hash table. This results in the hash table index that should be used to store the value.

Hash Functions for Strings

If the values are strings, we need some way to convert a string value to an integer. Once a string has been converted to an integer, we can mod it by the size of the hash table to determine the appropriate index at which to store the value.

Use all characters
Order matters (strings with same chars in different orders have different hash values)

One approach is to treat the binary representation of a string as a huge binary number. The problem with this approach is that the number represented by a string would easily overflow the range of the integer data types in any programming language. However, this isn't really a problem since we don't need to compute the huge number itself, but only the huge number mod the size of the hash table. This can be computed using an algorithm called "Horner's Method" that computes huge number mod table size without overflow.

Show code in HashFunctions.java.

Horner's Method has two disadvantages.

(1) Each loop iteration requires a mod operation which is expensive
(2) The resulting hash value depends only on the last few characters in the string. This means that all strings that end with the same characters will hash to the same index.

The second function in HashFunctions.java is an alternate hash function for strings that improves upon Horner's Method.

The third function in HashFunction.java is another hash function for strings.

Hash Functions for Objects

The values stored in a hash table could also be objects. As for strings, we need an algorithm for mapping objects to integers. Once we have the integer for an object, we only need to mod it by the hash table size to compute its hash function.

In fact, all Objects in Java have a method named `hashCode` that maps the target object to an integer. We can mod the integer returned by `hashCode` by the size of the hash table to determine the Object's index in a hash table.

The rule is that any two objects that are `equals` must return the same value from their `hashCode` methods.

How should we go about mapping objects to integers? For objects that contain multiple field values, the hash function needs to use the values of all field values that are used to distinguish one object from another. These are called **key fields**.

Show the Vehicle class in Vehicle.java.

For example, consider the `Vehicle` class. The hash function should be based on the `state` and `id` fields.

Collision Resolution By Chaining

The most common technique for dealing with collisions is called "**chaining**". When chaining is used, an entry in the hash table does not store a single value, but instead stores a linked list of all values that have hashed to that index. When we insert a new value into the hash table, we simply insert the value into the linked list at the appropriate index in the table. The list of values at a particular index is called a "**bucket**".

The bucket lists may be sorted to reduce the time required to search them.

Example: hash table with 10 slots, insert 89, 18, 49, 58, 9

What is the big-oh bound on the dictionary operations when chaining is used? Worst case is O(n), which occurs when all values hash to the same bucket. In the average case, the hash function evenly distributes the values throughout the table, and the bucket lists will have an average length of n/m, where m is the size of the hash table.

The ratio n/m is referred to as the "**load factor**" of the hash table. The higher the load factor, the worse the performance, so we want to keep the load factor low.

One advantage of chaining is that we do not have to know beforehand how many values will be inserted into the hash table. The chains grow as needed and can accommodate any number of values. Of course, our selection of the hash table size would benefit from knowing approximately how many values will be inserted, but this is not strictly necessary. As we shall see later, other techniques for dealing with collisions require prior knowledge of how many values will ultimately be inserted.

Open Addressing

We already discussed the chaining technique for handling collisions in a hash table. We now discuss another technique called "**open addressing**" for dealing with collisions.

With open addressing, we do not allow our hash table entries to store lists of values. Each table entry can store only one value. When we insert a new value, if the table entry for that value is already occupied, we must find another empty table entry in which we can store the value. This process of finding another open entry for the value is called "**probing**".

We can think of the probing process as a generalization of our hash function. In addition to the key argument, our new hash function takes a second argument that tells the function how many collisions we have had since we started trying to insert the key. For example, the first time call h(k, 0), which returns the first "**probe value**". If this entry is already full, we call h(k, 1), which returns another table entry to try. We call the hash functions as many times as necessary until we find an empty entry for the new value. This leads to the "**probe sequence**" [h(k, 0), h(k, 1), h(k, 2), …, h(k, m-1)]. We require that the probe sequence be a permutation of [0, 1, 2, …, m-1]. In other words, if we keep calling the hash function, we eventually visit each entry in the table, and we never visit the same entry twice.

What should the hash function look like when we use open addressing? We now discuss several alternatives.

Linear Probing

The simplest technique for probing the entries of the hash table is to simply move to the next entry in the table when the current entry is not available. When we reach the end of the array, we wrap around to the beginning of the array, thereby treating the array as a circular list.

Assume that we have a hash function h'(k) that is only a function of k. The hash function to use for linear probing looks like this:

$h(k, i) = (h'(k) + i) \bmod m$

Example: hash table with 10 slots, insert 89, 18, 49, 58, 9

Linear probing is easy to implement, but it suffers from a problem known as "**primary clustering**". Long runs of occupied slots build up, increasing the average search time.

Clusters arise since an empty slot that is preceded by x full slots gets filled next with probability (x + 1)/m. Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic Probing

Another approach that doesn't suffer from the primary clustering problem is called "**quadratic probing**". Quadratic probing uses a hash function that looks like this:

$h(k, i) = (h'(k) + i^2) \bmod m$

The offsets between the table entries visited by this function depend in a quadratic manner on the value of i. This avoids the primary clustering of linear probing,

Example: hash table with 10 slots, insert 89, 18, 49, 58, 9

When linear probing is used, it is clear that the probe sequence will eventually visit all table entries without revisiting any entries.

Is this still true for quadratic probing? To ensure that the probe sequence will eventually visit all table entries without revisiting any entries, we must ensure that the **hash table size is a prime number** and that the **load factor is no larger than 0.5**. [Note that in the example the hash table size was poorly selected because it wasn't prime.]

If two keys have the same initial probe, then they will always visit the same table entries in their probe sequence. This leads to a milder form of clustering called "**secondary clustering**".

Double Hashing

Another technique called "**double hashing**" is perhaps the best method for implementing open addressing. Double hashing avoids the clustering problems of linear probing and quadratic probing by using a hash function of the form:

$h(k, i) = (h1(k) + ih2(k)) \bmod m$

For any given key, the offset between the entries visited by the probe sequence is a constant number, but that number is computed based on the value of the key. Keys that have the same initial probe will not use the same offset, thus causing them to follow different probe sequences. This avoids the secondary clustering of quadratic probing.

When using double hashing, we must ensure that the value of h2(k) and m are relatively prime (i.e., they have no common factors). This ensures that the probe sequence will eventually visit all entries in the table. A convenient way to ensure this condition is to always have m be a prime number, and design h2 to always return a number less than m. For example, assuming that m is prime, we could define h1 and h2 as follows:

$h1(k) = k \bmod m$

$h2(k) = k \bmod (m-1) + 1$

Example: hash table with 10 slots, insert 89, 18, 49, 58, 9

Analysis of Open Address Hashing

When using open addressing, deletion of values from the hash table requires special care. When a value is removed from the table, the freed slot cannot be marked as "**empty**". This would cause searches for values whose probe sequences visited this entry when it was full to fail. One way to solve this problem is to mark the freed slot as "deleted", thereby making it available for new insertions, and also preserving the probe sequences of other values.

One disadvantage of open addressing compared to chaining is that you need to know approximately how many values will be inserted into the table beforehand. This prior knowledge may be difficult or impossible to acquire in some circumstances. Also, open addressing requires a bigger hash table than chaining, and the entire large table must be allocated up front.

For chaining, there was no requirement to ensure that the size of the hash table was greater than the number of values stored in the table. With open addressing, however, we must always ensure that $n < m$ (i.e., the load factor must always be less than 1.0).

Mathematical analysis has shown that the performance of open addressing severely degrades as the load factor n/m approaches 1.0. A good rule-of-thumb is to keep the load factor below 0.75. [For quadratic probing the load factor must be no larger than 0.5 in order to ensure a correct probe sequence.]

Rehashing

We just saw that for open addressing it is critical to keep the load factor well below 1.0. The load factor also affects the performance of chaining since it determines the average length of each bucket list. There is no requirement to keep the load factor below 1.0, but we should still like to keep it small to ensure good performance.

For both chaining and open addressing, we can use a technique called "rehashing" to effectively achieve O(1) average time performance for all of the dictionary operations. With rehashing, we select a target load factor that should never be exceeded. Then, if an insertion operation causes the load factor to exceed our specified limit, we create a bigger hash table and transfer all of the entries from the old hash table to the new one by rehashing each value.

Each value will almost surely hash to a different index in the new table, but that is not a problem.

Frequently, in computing there is a tradeoff between time and space. The faster we want something to run, the more memory it takes. Hashing is a classic example of a time/space tradeoff. Hash tables use more memory than is strictly necessary to store the

values that are in the table, but the extra memory allows us to achieve, in effect, O(1) performance for the dictionary operations. In fact, the most expensive operations in hashing are usually:

(1) initialization of the hash table array
(2) rehashing when the array needs to grow

Hashing Project

Go over the specification for the Hashing project.