Hashing

Motivation

As you’ve undoubtedly already realized, many applications call for (large) dynamic sets that require support for the basic dictionary operations SEARCH(X), INSERT(X) and DELETE(X). As a concrete example, imagine that we had the set of all Penn students’ names and we wanted quick lookups to see if a name existed. We might turn to a linked list, but lookup is $O(n)$. A better choice might be a balanced BST with student names as keys. This would give us $O(lg n)$ time lookups and insertions, which is already pretty good!

But say we’re going to be doing a lot of repeated lookups, so we’d like lookups on the order of constant time. How could we achieve this?

The Theory of Hashing

Direct Addressing

The first intuitive idea is to leverage the constant time lookup property of arrays. We can define a universe $U$ of possible keys (which in this case is all possible strings up to size $l$, where $l$ is the length of the longest name) and keep an array $A$ of size $|U|$. Then we can define a one-to-one mapping $M$ where we assign every string in $U$ to a number from 1 to $|U|$ and keep every name $k$ in the location $A[M(k)]$.

This method would give us guaranteed $O(1)$ lookups and inserts, but would take space $\Theta(|U|)$ which can be impractically large. For example, assuming that the longest name at Penn is 20 characters long (including spaces), there would be $27^{20}$ strings in our universe. This is clearly inefficient because there is no way we are going to be storing that many names.

Thus, Direct Addressing could be a good option when $U$ is small. But as $U$ becomes very large, storing an array of size $|U|$ can be impractical due to memory limitations. Furthermore, often the set of actual keys $K$ stored is much less than $|U|$, as such the extra space allocated to $A$ would be wasted.

Hash Tables

Instead of having one-to-one mappings from keys to slots in our array (referred to more generally as “table”) of size $|U|$, we can define a deterministic hash function $h$ that maps each key to some slot in our table $T[0...m − 1]$ of size $m$. Formally, the hash function $h$ is:

$$h : U \rightarrow \{0, ..., m − 1\}$$
where usually $m << |U|$.

By initializing a table of size $m << |U|$ and defining the hash function $h$, we’ve managed to save a lot of space. However, this improvement (like many things in life), comes at a cost: insertions and lookups are now on average $O(1)$ — worst case for some particular insert or lookup may be $O(n)$ — instead of guaranteed $O(1)$. Furthermore, by the pigeonhole principle, two keys in $U$ may now hash to the same slot, a situation known as a collision. Separate chaining and Open Addressing are two common ways of resolving collisions, which we’ll explore more in depth later.

N.B.: Our opening example used student names (which are strings) as keys. It should be noted that it is not difficult to convert any key to some non negative integer (see discussion questions). As such, in the rest of our discussions we assume keys are non-negative integers.

A brief note on the hash function

Let’s take a brief moment to consider what might make a good hash function. Consider the hash function $h(k) = k \mod 2$. We quickly realize that this is a horrible hash function — regardless of how large our symbol table is, we will have repeated collisions at slots 0 and 1.

It intuitively follows that we seek a hash function that distributes the universe $U$ of keys as evenly as possible among the $m$ slots of our table. We can formalize this concept and give it a name:

| Simple Uniform Hashing Assumption: Under this assumption, any key $k$ is equally likely to be mapped to any of the $m$ slots in our hash table $T$, independently of where any other key is hashed to. In other words, the probability of hashing some key $k$ not already present in the hash table into any arbitrary slot in $T$ is $\frac{1}{m}$ |

A direct result of this assumption is that if we would like to store $n$ elements in a table of size $m$, each slot will roughly get the same number $\left(\frac{n}{m}\right)$ of collisions. This is also known as the load factor, commonly denoted as by $\alpha$, of the table. The simple uniform assumption, however unrealistic, is used in the running

---

1To be more rigorous, we can define a random variable $X$ denoting the total number of elements we have hashed into $T$ and
time analysis of different collision resolution schemes, so make sure to keep it in mind.

The simple universal assumption, is afterall, just an assumption. In practice we rarely know the probability distribution from which the keys are drawn, and so we employ various heuristics when designing hash functions to achieve approximately an even distribution.

Furthermore, here’s something cool. Suppose that in our implementation of HashTable we always use the same hash function for every instance of our table. This is quite dangerous because if an adversary got hold of our hash function, he or she could work out what keys collide in our hash function and thus devise a dataset that would slow down our code and bring our system to a halt. The attacker can actually exploit the fact that insertions and lookups for hashtables are worst case $O(n)$ — say when all elements hash to the same slot — to start a denial of service attack on your application.

Thus in practice we often introduce randomization techniques to alleviate this worst case behavior. We maintain a carefully selected set of hash functions called a hash family, and whenever a new instance of our table is initialized, we choose a hash function uniformly at random from this set of hash functions to use for that specific table. Since the adversary doesn’t know which hash function we’ll use before hand, he or she cannot engineer a specific dataset that takes advantage of any particular hash function. The latest version of Java does this automatically. Note that each hash function $h$ in the hash family is still deterministic — that is, for a particular key $k$ and hash function $h$, $h(k)$ always has to produce the same result. The randomization comes from which hash function we choose to use when we initialize a new Hash Table. Once a hash function is chosen uniformly at random for an instance, it is used for the duration of the instance. There is a precise notion as to how to carefully select this set of hash functions (universal family) to consistently give good average case behavior.

Collision Resolution: Separate Chaining

The most obvious way to handle collisions is with a linked list. In other words, we chain all the keys that are hashed to slot $i$ in a linked list and store that list at slot $i$ in $T$. To find an element hashed to slot $i$, we would simply traverse the linked list until we found it (in the case of a successful search) or until we reached the end of the list (in the case of an unsuccessful search). Because of the simple uniform hashing assumption, the keys are roughly equally distributed amongst the $m$ linked lists in our hash table. With some math, we can show that on average, the number of elements examined in a successful search is $\Theta(1 + \alpha)$ where $\alpha$ is the load factor (Theorem 1). The number of elements examined in an unsuccessful search is $\Theta(1 + \alpha)$ as well (Theorem 2). If $\alpha$ is small, we can say that this is $O(1)$. However, we do not always know what $\alpha$ is at any given moment—thus we say that we can perform insertions and searches in expected $O(1)$ time.

$X_i$ is the number of elements that have been hashed to slot $i$.

$$E[X_i] = \sum_{j=1}^{n} \Pr[\text{element } j \text{ hashed to slot } i] = \sum_{j=1}^{n} \Pr[h(k_j) = i]$$

$$= \sum_{j=1}^{n} \frac{1}{m} \quad \text{(by Simple Uniform Hashing)}$$

$$= \frac{n}{m}$$

As a check, we can use Linearity of Expectation to solve for $E[X]$:

$$E[X] = \sum_{i=1}^{m} E[X_i]$$

$$= \sum_{i=1}^{m} \frac{n}{m}$$

$$= n.$$
Collision Resolution: Open Addressing

Preface: In this discussion, we will not consider deletions, as they can be quite tricky to implement with open addressing. As such, chaining is more commonly used when support for deletion of keys is necessary.

There are other ways besides chaining to handle collisions, and most of these strategies fall under the broad title of open addressing. In the technique of open addressing, instead of storing values that map to the same slot in a linked list, we find the “next” open spot in the same hash table to store our value. A consequence of this is that the hash table may become full, and the load factor $\alpha = \frac{n}{m}$ will always be less than or equal to 1.

When inserting a new element, we successively examine, or probe, the hash table according to some rule until we find an open spot to put our new element. We can define our probe sequence as $S = \{h_0(x), h_1(x), h_2(x), \ldots, h_{m-1}(x)\}$, where $h_i(x)$ represents the ith index we examine. We also say $h_0(x) = h(x)$. Defining $h_i(x)$ gives us the pre specified rules as to what index to check next—otherwise, we would just be choosing randomly. Note that you can make up your own rules as to how $h_i(x)$ is defined, but we need to make sure that the probe sequence is a permutation of $\{0 \ldots m - 1\}$ to ensure that eventually all slots are considered when inserting a new element as the table fills up. As we will see, there are different ways to specify our probe sequence, some yielding better results than others.

This pseudocode below for search (which is very close to insert) illustrates how open addressing works. Note that if an empty slot is found when searching for an element $x$, we can conclude that element doesn’t exist (assuming no deletions), since otherwise it would’ve been placed into that slot during insertion. The expected time for an unsuccessful search with this setup is $O\left(\frac{1}{1-\alpha}\right)$, which is $O(1)$ for values of $\alpha$ not near 1. [Theorem 3]. The expected number of probes for a successful search is harder to derive, and is given by $O\left(\frac{\pi}{2} \ln \frac{1}{1-\alpha}\right)$, which is also $O(1)$ when $\alpha$ is small.

Probing Techniques

There are variety of ways to define our probing function $h_i(x)$. In linear probing, we define our $h_i(x) = (h(x) + i) \mod m$. This has us taking equal sized steps through the table. While quick when the load factor is small (due to caching effects), as the load factor becomes large, insertion() and search() can end up traversing the entire table. This is due to a problem called primary clustering. This is just
function search(x):
    index ← h(x)  # Finds x in our hash table
    for i ← 1 to m do  # Sequentially probe the hash table
        if T[index] = x then  # We found x!
            return true
        elseif T[index] = ∅ then  # If encounter empty slot during probing, x doesn’t exist
            return false
        end if
        index ← hi(index)  # Compute the next index to “probe”
    end for
    return false  # Repeat unless entire probe sequence is complete
end function

what it sounds like. Because the difference between consecutive probes is linear, elements tend to form clusters. This is because once just a few elements are placed next to each other in the table, it becomes increasingly likely that an element will hash to one of the indices occupied by an element in that cluster. Then, because linear probing is used, that element will be placed near the cluster once it finally finds an open address, effectively increasing the size of the cluster. This can be mitigated by other forms of probing such as quadratic probing: \( h_i(x) = (h(x) + i^2) \mod m \). However, quadratic probing suffers from a milder yet similar problem called secondary clustering. An even more robust form of open addressing is double hashing (where we choose another hashing function \( h'(x) \) so \( h_i(x) = (h(x) + i \cdot h'(x)) \mod m \)). This makes the probe sequence vary depending on the element being inserted instead of the initial probe index, effectively eliminating clustering.

![Figure 3: Insertion of \( k = 14 \) with double hashing, with \( h(k) = k \mod 13 \) and \( h'(k) = 1 + (k \mod 11) \). Hash table is size 13 so \( m = 13 \). \( h(14) = 14 \mod 13 = 1 \) and \( h'(14) = 1 + (14 \mod 11) = 4 \). Hence procedure inserts 14 into slot 9 after probing slots 1 and 5 (recall \( i \) begins at 0).](image)

If at this point you’re feeling a little lost as to how probing works, don’t worry — the formalization makes it seem confusing but working out a few simple examples make linear and quadratic probing as well as double hashing much clearer. Remember, these probing techniques are no more than “rules”, which we are even free to make up ourselves, to specify the next index to check!

**Hash Maps in the Real World (for fun, 100% optional)**

The debate between chaining and open addressing does not have a clear-cut winner; there are merits to both solutions and both are widely used in the real world. For example, Java provides both implementations: `java.util.HashMap` uses chaining while `java.util.IdentityHashMap` uses linear probing. Python has chosen to
use open addressing with random probing.\(^2\)

Since we are working with Java, however, we will analyze implementation of `java.util.HashMap` (IdentityHashMap is not commonly used).

**Java HashMap**

Java's `HashMap` constructor allows the user to define an initial capacity and a load factor. `java.util.HashMap` is implemented with separate chaining. The default capacity is 16, and the default load factor is 0.75. Here the load factor is defined as \( \frac{n}{m} \), where \( m \) is the size of the table and \( n \) is the number of elements that can be inserted before the table should be resized. Java rehashes all elements when resizing the underlying table.

**Testing**

We will vary initial capacity and load factor values and look at some performance results.

In our test program, we call `put` on a million random numbers, then we call `get` on the same million numbers. We run this test 100 times to eliminate the possibility of bad sets of numbers or CPU usage spikes affecting the result. The average times are then recorded (in milliseconds) for the `put` and `get` operations.

The following table varies initial capacity and uses the default load factor 0.75.

<table>
<thead>
<tr>
<th>Initial Capacity</th>
<th>put</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^4 )</td>
<td>194.88</td>
<td>55.86</td>
</tr>
<tr>
<td>( 2^5 )</td>
<td>196.16</td>
<td>56.3</td>
</tr>
<tr>
<td>( 2^6 )</td>
<td>195.37</td>
<td>56.06</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( 2^{17} )</td>
<td>193.12</td>
<td>56.15</td>
</tr>
<tr>
<td>( 2^{18} )</td>
<td>188.83</td>
<td>55.26</td>
</tr>
<tr>
<td>( 2^{19} )</td>
<td>180.37</td>
<td>56.29</td>
</tr>
<tr>
<td>( 2^{20} )</td>
<td>142.55</td>
<td>54.31</td>
</tr>
<tr>
<td>( 2^{21} )</td>
<td>69.26</td>
<td>54.19</td>
</tr>
<tr>
<td>( 2^{22} )</td>
<td>64.58</td>
<td>48.94</td>
</tr>
<tr>
<td>( 2^{23} )</td>
<td>63.04</td>
<td>45.72</td>
</tr>
<tr>
<td>( 2^{24} )</td>
<td>63.17</td>
<td>44.89</td>
</tr>
</tbody>
</table>

We can see that `put` times are very stable until we get close to the 1 million mark, at which point they decrease dramatically, and stabilize after that. This is due to the HashMap rehashing many times. For initial capacities of \( 2^{21} \) or greater, no rehashing ever occurs. Setting initial capacity appropriately is very important to the running time of HashMaps.

You may have noticed that `get` times decreased after the \( 2^{21} \) point as well. This is because the table is sparse so there are less collisions on average.

Next we look at varying load factor, with an initial capacity of 16 (default).

<table>
<thead>
<tr>
<th>Load Factor</th>
<th>put</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>284.05</td>
<td>51.22</td>
</tr>
<tr>
<td>0.45</td>
<td>225.25</td>
<td>51.72</td>
</tr>
<tr>
<td>0.75</td>
<td>194.38</td>
<td>54.44</td>
</tr>
<tr>
<td>1</td>
<td>154.88</td>
<td>66.18</td>
</tr>
<tr>
<td>2</td>
<td>177.19</td>
<td>85.01</td>
</tr>
<tr>
<td>5</td>
<td>314.93</td>
<td>125.4</td>
</tr>
</tbody>
</table>

We can see here why HashMap has a default load factor of 0.75. Load factors smaller than 0.75 offer insignificant `get` improvements, while causing lots of table space to be wasted. On the other hand, load factors greater than 0.75 do in fact cause a slow down of `get` operations.

\(^2\)There is a good discussion of Python's implementation [here](https://docs.python.org/3/tutorial/datastructures.html#dictionaries).
Worst Case Linear Time

With the correct load factor and initial capacity, we can observe the expected constant time access of HashMaps. However, for any hash function, there exists a set of inputs for which performance degrades to linear time. The following table shows inserting into a HashMap a set of values that are known to collide.

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^1$</td>
<td>4</td>
</tr>
<tr>
<td>$2^2$</td>
<td>9</td>
</tr>
<tr>
<td>$2^3$</td>
<td>19</td>
</tr>
<tr>
<td>$2^4$</td>
<td>36</td>
</tr>
<tr>
<td>$2^5$</td>
<td>73</td>
</tr>
<tr>
<td>$2^6$</td>
<td>155</td>
</tr>
<tr>
<td>$2^7$</td>
<td>296</td>
</tr>
<tr>
<td>$2^8$</td>
<td>742</td>
</tr>
<tr>
<td>$2^9$</td>
<td>1617</td>
</tr>
</tbody>
</table>

As mentioned before, the way to circumvent this is to pick a random hash function every time a new HashMap is created. The latest version of Java implements this.

Discussion

Chaining vs. Open Addressing

Compare chaining and open addressing and discuss the pros and cons of each.

Solution

Chaining Pros:
- relatively simple to implement

Chaining Cons:
- can have long chains
- overhead from having to traverse pointers and maintain linked lists

Open Addressing Pros:
- takes up less space than chaining
- linear probing can take advantage of caching

Open Addressing Cons:
- clustering
- deletions can be tricky
- can have long probe sequences

Methods of chaining

What effects would it have if, instead of chaining with linked lists, you used balanced binary search trees? What about if you used hash tables (in a sort of recursive fashion)?
Solution
Implementing chaining with balanced binary search trees (so that the height of the BST is always \( \lg n \)) would give order to the data at each index, allowing us to have worst case \( O(\lg n) \) time lookups from our hash table as opposed to worst case \( O(n) \) time lookups if we used a linked list. However, we would then have increased overhead and memory usage from maintaining the balanced binary search tree. Furthermore, the worst case \( O(n) \) from the linked list is mitigated by keeping the table size large, which allows lookups to be expected constant time.

If you chain with hash tables, you could theoretically end up having a large recursive structure of hash tables (essentially like a linked list of hash tables), which would use a lot of memory and take a lot overhead to navigate, so this is not ideal. It would still be worst case \( O(n) \) time for lookups, but in practice would likely be slower due to the aforementioned overhead.

Methods of Open Addressing
Compare linear probing and double hashing. What problems do each suffer from? Which is better?

Solution
Linear probing suffers from primary clustering, so it is generally worse than double hashing. Something that linear probing has over double hashing is its ability to make use of data caching. It can do this because the distance between successive probes in a sequence is relatively small. Contrastingly, the distances between successive probes in double hashing are, on average, larger.

Designing Hash Codes
We saw that arrays require integer indices, and that most hash functions assume the universe of keys is the set of natural numbers. Hence if the keys are not natural numbers, we need to find a way to interpret them as such. How would you do so for the following keys?
(Note hashCodes are not the output of hash functions, but are used by the hash function to determine which bucket to hash your input to)

1. A boolean
2. A student name (“String”)
3. A point object storing the coordinates \((x, y)\)
4. A complex object containing 100 fields and various public and private methods.

Solution
1. If true then 0 else 1.
2. Intuitively each character in the string has an ASCII integer value, so the most naive implementation is to simply take the sum of all the ASCII integer values. In practice, however, this does not create a very even distribution of keys, since as a simple example the strings “abc” and “cba” would produce the same hash codes under this scheme. This naturally leads to the idea that we also need to take the position of characters into account, and perform some weighted average over the characters of the string.

In fact, since Java 1.2, java.lang.String implementation of computing hashcodes for strings is:

\[
h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}
\]

where \( s \) is the input string and \( s[i] \) denotes the UTF-16 code of the i-th character of \( s \), and \( n \) is the length of \( s \).
3. Since \(x\) and \(y\) both contain information about the point object, it makes sense to incorporate both when computing the \texttt{hashCode}. If they are both doubles, we can convert them to longs using \texttt{doubleToLongBits(double value)} which return a representation of the double according to IEEE 754 floating point specifications. We then might simply add them together. In practice this does not produce a even distribution of hash codes so Josh Bloch’s Effective Java actually suggests calculating the hash code for each long \(f\) as \((\texttt{int} f \ ^{\ (f \gg 32)}\), where \(\gg\) is the logical right shift operator and \(^\). The final hash code combines the two pieces as \texttt{hashCode = 37 * hashCode(x) + hashCode(y)}. 37 is a rather arbitrary choice, though empirically this gives a good distribution of hashCodes, though the mathematics is beyond this discussion. Note that computing hash codes generally involves prime numbers.

4. As we quickly realize, computing hashCodes of an object involves using the information from a subset of its fields. If the object has a unique id, for example, we can just use the hashCode of the id for the entire object. Otherwise we can compute the hashCodes of every field used in the \texttt{equals method}, say with the methods suggested \texttt{here}, and combine all the hashCodes as \texttt{result = 37 * result + c}, where \texttt{result} was initially set at 0 and \(c\) is the hashCode of the next field.

Alternatively, one might consider using the object’s memory address compute its hashCode.

There are, of course, many ways to implement hashCodes. This is just to provide a flavor for how it might be done, especially by java, in the real world.

**Problems**

**Problem 1**

You are given an array \(A\) containing distinct randomly assorted integers. Your goal is to find two elements in the array whose sum is \(k\) in \(O(n)\) expected time.

**Solution**

Begin by traversing \(A\) and putting all of the elements in a hash table. Then traverse \(A\) again, and for each element \(e\), check and see if \(k - e\) exists in the hash table (and make sure \(k - e \neq e\)). If it does, simply return \(e\) and \(k - e\). If you reach the end of the array and haven’t found a pair, return that no such pair exists.

Running time Analysis: It takes expected \(O(n)\) time to insert all of the elements of \(A\) into the hash table, \(O(n)\) time to traverse \(A\) again, and expected \(O(1)\) time for each of the \(n\) elements (yielding expected \(O(n)\) time) to search for its complement. Therefore our algorithm still runs in expected \(O(n)\) time.

**Problem 2**

Assume we have a hash table \(T\) of size 10 that uses linear probing and has hash function \(h(x) = x \mod 10\). We insert 6 numbers into \(T\) and we get the below table:

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
What is one possible order that we could have inserted these elements to get this result?

How many probes would be required for inserting 13 in the table?

Solution

One possible order that the numbers could have been inserted in is 46, 34, 42, 23, 52, 33. For a sequence to be valid:

- 46 must be inserted before 33
- 42, 34, and 23 must be inserted before 33 and 52

It would take 6 probes to insert 13 into the table, as we would first try to insert it in index 3, fail, and then traverse the cluster until we reach index 8, at which point it would be inserted after we realize that $T[8]$ is empty. Note that this is a prime demonstration of primary clustering.

Problem 3

How would you detect a cycle in a linked list of distinct elements in expected $O(n)$ time?

Can you do it in constant space?

Solution

Our algorithm is as follows: Traverse the linked list, and keep a hash table of every element we’ve encountered. As we reach a new element in the linked list, check to see if it already exists in our hash table. If it does, we’ve found a cycle. If we reach the end of the linked list, then we know that no cycle exists.

Running time Analysis: It takes linear time to traverse the linked list and expected $O(1)$ time for each lookup and insertion in the hash table, of which there could be $2n$. Thus, the algorithm is expected $O(n)$ time.

To do it in constant space, we can begin with two pointers $i$ and $j$ and traverse the array. To traverse the array, we move the pointer $j$ two spots along the linked list for every spot we move $i$. If there is a cycle, we know that $j$ will eventually intersect with $i$ (think about why this is!). If there is no cycle, then $j$ will reach the end of the linked list and we are done.

Running time Analysis: We can say this algorithm is $O(n)$—try to figure out why this is and see if you can come up with a more specific bound in the case of a cycle!

Problem 4

Design an algorithm that determines if two lowercase words are anagrams of each other in expected $O(n)$ time. Note: a string $A$ is an anagram of another string $B$ if $A$ is a permutation of $B$.

Can you do it in worst case $O(n)$ time?

Solution

Expected $O(n)$ time solution: First of all, if $A$ and $B$ aren’t the same length, they cannot be permutations. Otherwise, use two hash tables where the keys are letters and the values are the counts of those letters. Iterate through $A$ incrementing the count of a letter in $A$’s hash table every time that letter is seen in $A$. Similarly, iterate through $B$ incrementing the count of a letter in $B$’s hash table every time that letter is seen in $B$. This effectively builds a map that maps from every letter in $A$ to the number of times that letter appears in $A$. It also does the same for $B$. Then, iterate through the letters in $A$’s map, and see if their counts are the same as their counts in $B$’s map. If this passes, we know $A$ and $B$ are the same. Note that we didn’t have to iterate through the letters of $B$ because we know $A$ and $B$ are the same length.

Worst Case $O(n)$ time solution: Because there are only 26 lowercase letters in the alphabet, we can just use 2 arrays of size 26 each to store the counts. Then, it’s just the same algorithm as was used above except the keys above are just indices in the arrays and the values above are the values in the arrays at those.
indices. Because arrays support worst case \( O(1) \) time insertions and lookups, and we iterate through each string once, this implementation runs in worst case \( O(n) \) time.

Alternatively, you could just build the hash table/array the same way for string \( A \), then traverse string \( B \) and for each letter, decrement the corresponding counter in \( A \)'s map/array. If a counter ever becomes \(-1\) or the letter did not exist in \( A \), return false. Otherwise, return true (assuming we already checked if \( A \) and \( B \) are the same length).

### Auxiliary proofs (mathematical details for analysis of hashing)

**Theorem 1. (Chaining successful lookup)** In a hash table where collisions are resolved by separate chaining with newly colliding elements added to the front of the chain, the average number of elements examined in a successful search is \( \Theta(1 + \alpha) \), where \( \alpha = n/m \) is the load factor, \( n \) is the number of elements in the hash table, and \( m \) is the number of buckets.

**Proof.** Let \( k_1, k_2, \ldots, k_n \) be the elements of the hash table in order of insertion. Assume that for any particular search, any element is equally likely to be the element we are attempting to look up.

Let \( Z \) be the random variable representing the number of elements examined in a successful search. If we compute \( E[Z] \), we are done.

Define an indicator random variable

\[
X_{ij} = \begin{cases} 
1 & \text{if } h(k_i) = h(k_j), \\
0 & \text{otherwise}.
\end{cases}
\]

By *simple uniform hashing*, \( \Pr[h(k_i) = h(k_j)] = \frac{1}{m} \), so

\[
E[X_{ij}] = \Pr[X_{ij} = 1] = \Pr[h(k_i) = h(k_j)] = \frac{1}{m}
\]

Recall that colliding elements are added to the front of a chain. An element \( k_i \) only has \( k_{i+1}, k_{i+2}, \ldots, k_n \) in front of it in the chain. Then the number of elements that are examined to find to find a particular \( k_i \) (including considering \( k_i \) itself) is simply

\[
Z_{k_i} = 1 + \sum_{j=i+1}^{n} X_{ij}.
\]

In expectation, we have

\[
E[Z_{k_i}] = E \left[ 1 + \sum_{j=i+1}^{n} X_{ij} \right] = 1 + \sum_{j=i+1}^{n} E[X_{ij}]
\]

\[
= 1 + \sum_{j=i+1}^{n} \frac{1}{m} = 1 + \frac{n - i}{m}
\]
Since each $k_i$ is equally likely to be searched for, we take the average over all possible $Z_{k_i}$ to find $E[Z]$:

\[
E[Z] = \sum_{i=1}^{n} \frac{1}{n} E[Z_{k_i}]
\]

\[
= \sum_{i=1}^{n} \frac{1}{n} \left(1 + \frac{n-i}{m}\right)
\]

\[
= \sum_{i=1}^{n} \left(\frac{1}{n} + \frac{1}{m} - \frac{i}{nm}\right)
\]

\[
= 1 + \frac{n}{m} - \frac{1}{nm} \sum_{i=1}^{n} i
\]

\[
= 1 + \frac{n}{m} - \frac{1}{nm} \cdot \frac{n(n+1)}{2}
\]

\[
= 1 + \frac{n}{m} - \frac{n+1}{2m}
\]

\[
= 1 + \frac{n-1}{2m}
\]

\[
= 1 + \frac{n}{2m} - \frac{1}{2m}
\]

\[
= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n},
\]

so $E[Z] = \Theta(1 + \alpha)$.

\[\blacklozenge\]

**Theorem 2. (Chaining unsuccessful lookup)** In a hash table where collisions are resolved by separate chaining with newly colliding elements added to the front of the chain, the average number of elements examined in an unsuccessful search is $\Theta(1 + \alpha)$, where $\alpha = n/m$ is the load factor, $n$ is the number of elements in the hash table, and $m$ is the number of buckets.

**Proof.** Under the assumption of simple uniform hashing, any key $k$ not already stored in the table is equally likely to hash to any of the $m$ slots. The expected time to search unsuccessfully for a key $k$ is the expected time to search to the end of list. The expected length of the chain is exactly equal to the load factor $\alpha$, so the total time required is $\Theta(1 + \alpha)$.

**Theorem 3. (Open addressing unsuccessful lookup)** In a hash table where collisions are resolved by open addressing, the average number of elements examined in an unsuccessful search is $O\left(\frac{1}{1-\alpha}\right)$.

**Proof.** We can show this by writing the expected number of probes as a recurrence on the two variables $m$ and $n$:

\[
E[T(m, n)] = 1 + \frac{n}{m} E[T(m-1, n-1)]
\]

The $\frac{n}{m}$ term arises from the expected number of collisions.

We can prove using induction that $E[T(m, n)] \leq \frac{m}{m-n} = \frac{1}{1-\alpha}$.

**Base Case:** The base case arises at $T(m, 0)$, which means that the hash table is empty. Thus the first probe always finds an empty element, so $T(m, 0) = 1$.

**Induction hypothesis:** $E[T(m-1, n-1)] \leq \frac{m-1}{m-n}$.
Induction step:

\[ E[T(m, n)] = 1 + \frac{n}{m} E[T(m - 1, n - 1)] \]

\[ \leq 1 + \frac{n}{m} \cdot \frac{m - 1}{m - n} \]

\[ < 1 + \frac{n}{m} \cdot \frac{m}{m - n} \]

\[ = \frac{m - n}{m - n} + \frac{n}{m - n} \]

\[ = \frac{m}{m - n} = \frac{1}{1 - \alpha} \]

\[ \therefore E[T(m, n)] \leq \frac{1}{1 - \alpha} \]