Chapter 17
Hash Tables and Collision Resolution

17.1 The Dictionary ADT

Suppose you wanted to write a program for a local grocery store that, when a user enters in a shopping item, returns the price of the item that was entered. When the program begins, it loads in a data set containing the prices of all the items in the store by item name. How would you store this data in your program to support this lookup?

Food Item	Price
Apple	$3.99
Avocado	$4.99
Banana	$2.49
Flour	$2.29
Ginger	$2.69
Milk	$2.09
Tofu	$3.79

User queries: `"Tofu"`
Program returns: `3.79`

One way to accomplish this is to use an abstract data type known as a **dictionary**. A dictionary is a lookup container comprised of a collection of key-value pairs, where the *key* is an object you can query information for, and the *value* is the data associated with a given key. In most cases, each key in a dictionary is unique. By utilizing key-value pairs, each key of a dictionary is "mapped" to a corresponding value that can be queried using the key's value. This makes a dictionary a good container type if you want to look up information associated with a given object.

Remark: The *dictionary* moniker is quite fitting for this abstract data type, since its behavior closely resembles that of an actual dictionary! When you use a dictionary, you are taking a word and looking up its definition. Here, the "key" is the word you want to look up, and the "value" is the definition of the word. The dictionary abstract data type (ADT) applies this idea to data retrieval in a program: if you use a dictionary ADT, you can look up information associated with any object by storing this lookup data in the form of key-value pairs.

Using the shopping example from earlier, we could use a dictionary ADT to support lookup for the price of any item given its name. In the dictionary, we would store the item-price relationships as key-value pairs, where the name of the item is the key (the object that you want to look up data for), and the price is the value (the information associated with each key).

Food Item	Price
Apple	$3.99
Avocado	$4.99
Banana	$2.49
Flour	$2.29
Ginger	$2.69
Milk	$2.09
Tofu	$3.79

There are several basic operations that a dictionary should be able to support:

- the insertion of new key-value pairs
- the retrieval (or lookup) of values associated with a given key

It should be noted that a dictionary is an *abstract data type*, so it is simply an *interface* for a container that supports key-value lookup. As mentioned, if you have a dictionary, you can insert key-value pairs, and you can look up the value associated with any key. However, the actual implementation details of a dictionary can vary greatly based on the features that a specific lookup container should provide. A lookup container that allows sorted access to its key-value pairs is implemented very differently from a container that does not need to exhibit this behavior, even though both are considered as dictionaries as long as they support the dictionary interface.

As such, our exploration of the dictionary ADT will be split into two chapters. In this chapter, we will discuss the implementation of a dictionary that does not need to store its key-value pairs in any particular order. This implementation is known as a *hash table* or a *hash map*. Because they do not need to maintain an internal ordering of elements, hash tables can be implemented to support key-value lookups in *constant* time. The fast performance of hash tables makes them very useful for solving a variety of problems that would be less efficient if implemented using other containers, and knowledge of this data structure in your coding repertoire will be immensely beneficial to you as a programmer.

However, there are situations where you may want a dictionary to support more than just key-value lookup. For instance, you may also want a lookup container to provide *sorted* access to its key-value pairs. To implement a dictionary that stores its keys in sorted order, we will use a structure known as a *binary search tree*. These sorted dictionaries are versatile and support more features than a standard, unordered hash table, but these features come at the expense of constant time lookup. We will discuss this implementation in greater detail in chapter 18.

17.2 Introduction to Hashing

※ 17.2.1 *Hash Tables*

Let's return to the shopping example introduced earlier in this chapter. How would you efficiently implement a dictionary that allows you to query the price of an item as quickly as possible? Using the data structures we have covered so far, this is not immediately apparent. We could try to store the values in an array-like container and loop through the container whenever a user queries an item. An example is shown below:

```
1  struct Item {
2    std::string name;
3    double price;
4  };
5
6  // loop through all items and find the one that matches the target
7  double get_price(const std::vector<Item>& items, const std::string& target) {
8    for (const Item& item : items) {
9      if (item.name == target) {
10       return item.price;
11     } // if
12   } // for
13   throw std::invalid_argument("Item does not exist!");
14 } // get_price()
```

However, this approach requires us to perform a $\Theta(n)$ traversal of all the items in our container whenever we want to query the price of a single item. This is not the best way to do things! Instead, we want to store the data in a way that supports efficient lookup: if we query any item, we should be able to get back its price without having to look at all other items in the container. How can this be done?

To gain some intuition to this better method, let's adjust the previous example a bit and replace each item's name with an ID number starting from zero, as shown below:

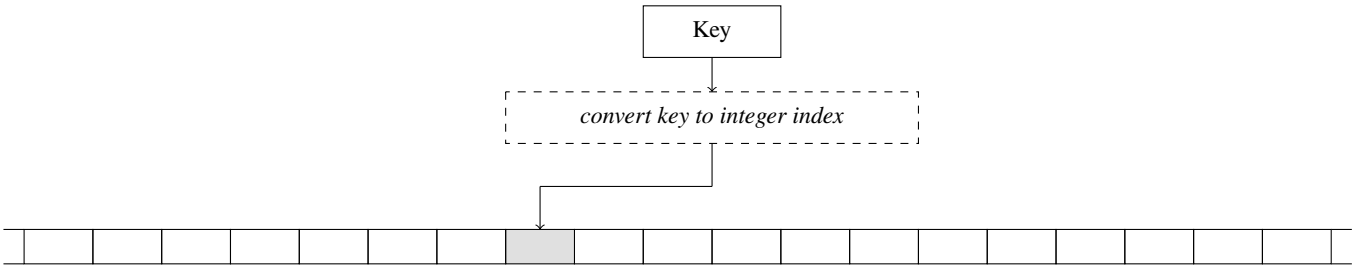
Food Item	Price
0	$3.99
1	$4.99
2	$2.49
3	$2.29
4	$2.69
5	$2.09
6	$3.79

How can we store this data in a way that allows us to efficiently retrieve a food item's price given its ID number? The straightforward solution would be to insert all the price values into an underlying data array, where the ID of an item is represented by each value's index:

3.99	4.99	2.49	2.29	2.69	2.09	3.79
0	1	2	3	4	5	6

If we were given an item number, we could just query the price of that item by using its item number as an index (e.g., `vec[1]` for the price of item 1). This provides lookup in constant time! Yet, this is not very different from the problem we had earlier; we simply replaced each item name with an integer ID instead. The main difference is that strings cannot be used to index directly into an array (e.g., `vec["Apple"]` does not work), but a numeric ID can, provided that it is non-negative.

Although strings cannot be used to directly index into an array, we can resolve this issue by simply converting each string into a numeric index that can be used to index an array. This concept applies to any key type as well: if we can convert a key into an integer index, then we can support lookup on that key by first converting it into an index, and then using that index to access a value from an underlying array (with some caveats of course, which will be discussed in this chapter). If this conversion takes constant time, then lookup can also be done in constant time.



use this converted index to index into an underlying array that stores the values for each key

This basic idea forms the foundation for the hash table data structure. A **hash table**, alternatively known as a **hash map**, is an implementation of the dictionary ADT that supports key-value insertion, lookup, and removal in average-case *constant* time with respect to the total number of items in the hash table. This constant time access is made possible using a process known as *hashing*, which is designed to translate a key into an index value that is then used to access an underlying array that stores the values associated with each key.

※ 17.2.2 *Hash and Compression Functions*

When a hash table lookup is requested on a key, the key is first passed into something known as a **hash function**, which performs arithmetic operations to convert that key into an integer. This integer is then compressed into a valid index, which is then used to index into an underlying data vector storing the hash table's key-value pairs. To illustrate this process, consider the following hash function that converts a string key into an integer based on its first letter. We will use this hash function to implement a hash table that stores its data in an underlying array of size 10.

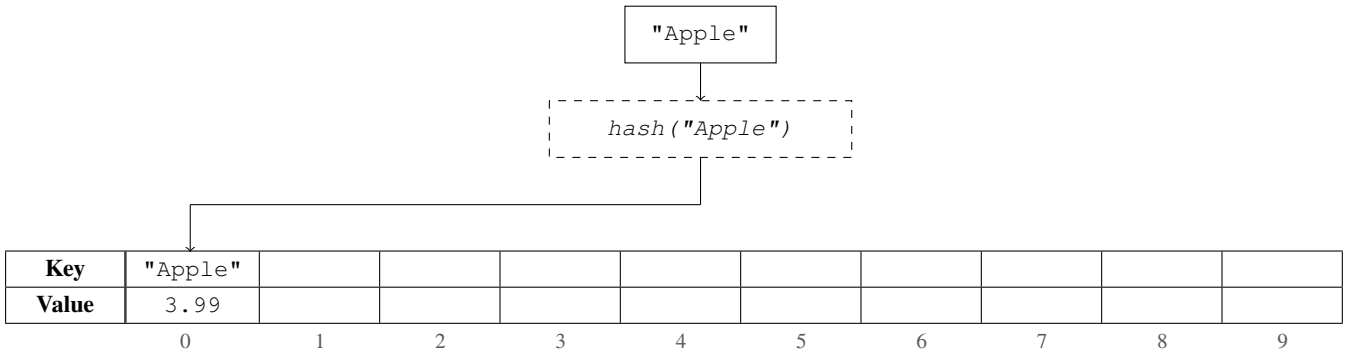
```
1  int32_t hash(const std::string& s) {
2    if (s.empty()) {
3      return 0;
4    } // if
5    // returns 0 if first char is 'A', 1 if 'B', 2 if 'C', ..., 25 if 'Z'
6    return (s[0] - 'A');
7  } // hash()
```

Key										
Value										
	0	1	2	3	4	5	6	7	8	9

Suppose we wanted to insert the following key-value pair into this hash table:

```
{"Apple", 3.99}
```

To determine which index of the underlying vector this key-value pair falls into, we will pass the key `"Apple"` into the hash function. Since `"Apple"` begins with the character `'A'`, the hash function returns 0, and the key-value pair `{"Apple", 3.99}` ends up at index 0. Here, we consider 0 as the **hash value** of the key `"Apple"`.

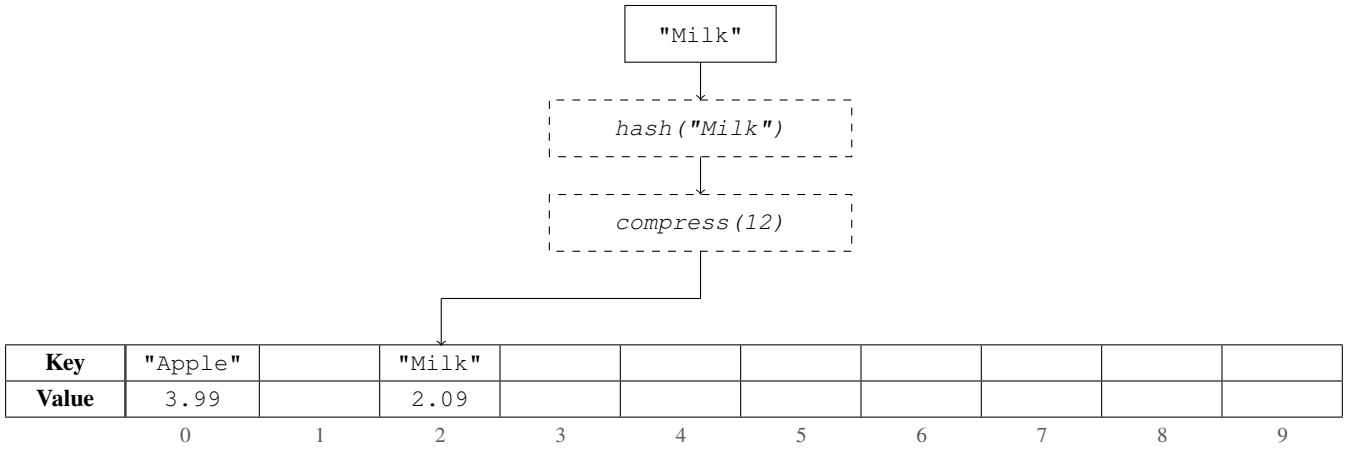


Now, suppose we wanted to insert the following key-value pair into this hash table:

```
{"Milk", 2.09}
```

If we were to pass `"Milk"` into our hash function, we would get back a hash value of 12. However, 12 is not a valid index number since our vector only has a size of 10. This is where the **compression** step plays an important role. During the compression step, the result obtained from the hash function is compressed into a valid index in the range $[0, M)$, where M is the size of the underlying array. The simplest compression method is to take the modulo of the hashed integer with the size of the table (e.g., $h(t) \bmod M$).[1] In this case, 12 mod 10 is 2, so `{"Milk", 2.09}` would end up at index 2.

```
1  int32_t compress(int32_t hash_result) {
2    return hash_result % table_size;
3  } // compress()
```



Now, suppose we wanted to retrieve the price of `"Apple"`. With hashing, we can do this in $\Theta(1)$ time: we first pass the string `"Apple"` into the hash function, which returns 0. Then, we go to index 0, confirm that the key is indeed `"Apple"`, and then return the corresponding value (`3.99`) to the user. Notice that, as long as the hash and compression functions take constant time, the entire process of finding the price of any item (and similarly inserting or removing items) also takes $\Theta(1)$ time. If we were to insert the remaining foods into the table (excluding `"Avocado"`, which produces a special scenario that we will cover later), we would get the following hash table:

Key	`"Apple"`	`"Banana"`	`"Milk"`			`"Flour"`	`"Ginger"`			`"Tofu"`
Value	`3.99`	`2.49`	`2.09`			`2.29`	`2.69`			`3.79`
	0	1	2	3	4	5	6	7	8	9

[1] If $h(t)$ can be negative, an absolute value should be applied to the hash value before taking the modulus.

※ 17.2.3 *Properties of Hash and Compression Functions*

In the previous example, we used a hash function that calculated a string's hash value using the distance between the first letter of the string and the letter `"A"`. However, this is not the only hash function we could have used. In fact, we could have used any function that converts a key into a numeric hash value, as long as it has the following properties:

- **It must be easy to compute.** If a hash function is so complicated that running it takes longer than constant time, it defeats the purpose of having a hash table in the first place.
- **It must compute a hash for every key.** The hash function should be able to convert any valid key into an integer. For instance, the hash function should not return integers for some keys and segfault for others.
- **It must compute the same hash for the same key.** The benefit of hashing comes from the fact that the index each key maps to is deterministic. If the hash function converts the string "Apple" to the integer 0, then it should always convert "Apple" to 0. It shouldn't occasionally convert "Apple" to something else, since that would prevent us from getting the efficient lookup we need from a hash table.

There is another property that a hash function *should* have to be able to provide consistent $\Theta(1)$ lookup. This property is not required for a hash function to be valid, but having it allows a hash table to perform its operations efficiently.

- **It should distribute keys evenly.** For instance, "Apple" and "Avocado" are hashed to the same location using the hash function in the previous example (we will deal with this issue in the next section). When two keys map to the same index, a *collision* occurs. If a collision happens, we will have to do additional work to resolve the collision, which may take more than $\Theta(1)$ time. There is no way to eliminate collisions entirely, but they can be minimized if keys are distributed evenly by the hash function.

For instance, a hash function that always returns 0 for all keys is a valid hash function (since it satisfies the first three mandatory requirements), but it is a very poor hash function since it causes collisions every time a key is inserted.

The compression step also plays a role in the efficiency of a hash table. There are two primary methods of compressing the output of the hash function $h(t)$ into an index in the range $[0, M)$, where M is the table size.

- The **Division Method**: $|h(t)| \bmod M$. Ideally, the table size M should be a prime number.
- The **MAD (Multiply and Divide) Method**: $|a * h(t) + b| \bmod M$, where a and b are prime numbers. This method can be used if you cannot control the underlying table size M. In this method, $a \bmod M$ must not equal 0.

If you could choose the table size M, why should M be prime? It turns out that prime number table sizes reduce collisions and improve the performance of a hash table. For example, if you created a hash table with a size of 100 and all your keys ended up hashing to multiples of 5, your hash function would fail to distribute keys evenly (e.g., no element would be mapped to indices 1-4, 6-9, …). However, if you changed the table size to 101 (a prime number), you would get a more even distribution of keys, even if every key were hashed to a multiple of 5.

In general, non-prime values of M tend to cause more collisions because every hash value that shares a common factor with M will end up at an index that is a multiple of this factor (which prevents the keys from being distributed evenly). However, if M were prime, you wouldn't have to worry about keys sharing a common factor with M (since the only factors of a prime number are 1 and itself).

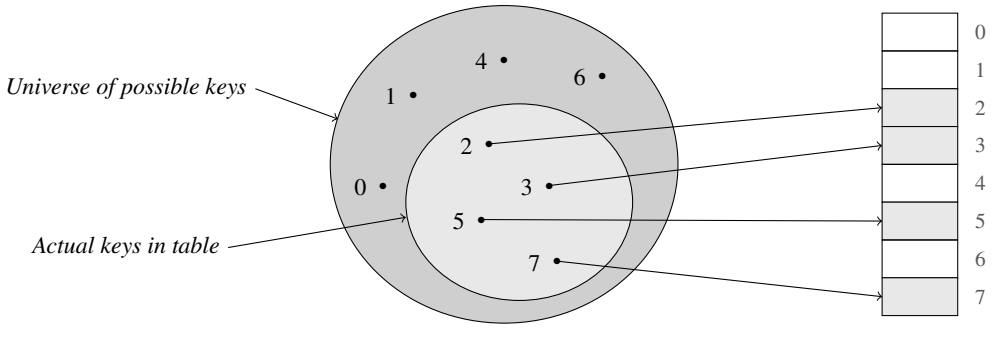
Claim: Every key K that shares a common factor with the table size M will be hashed to an index that is a multiple of this common factor.

Proof: Let K and M be natural numbers ($\in \mathbb{N}$) such that K and M share a common factor. Without loss of generality, let the natural number t be one of the common factors of K and M. Since K and M have a common factor t, we know that $K = tu$ and $M = tv$ for some natural numbers u and v. Furthermore, the key K will be hashed to some index i, where $i = K \bmod M$. Since i is the remainder when K is divided by M (per the definition of a modulus), there must exist an integer a such that $i = K - aM$. Replacing K with tu and M with tv, we get the equation $i = tu - atv$. This can be simplified to $i = t(u - av)$. We have now proven that the index i is a multiple of the common factor t. Thus, all keys that share a common factor with the table size will end up at an index that is a multiple of the common factor. Note that this holds for all common factors between K and M.

If you could guarantee perfect hashing with no collisions, what are the time complexities of insertion, search, and removal? In the case of insertion, if you can guarantee that every key had its own index, you would immediately know where each key should go in the hash table. Thus, insertion would always take $\Theta(1)$ time as long as the hash function also takes constant time. The same applies to search and removal — you know exactly where each key can be found in the hash table, so you can immediately retrieve its value or erase it in $\Theta(1)$ time. However, perfect hashing is a theoretical ideal that is hard to attain in practice. This is because the hashing process is rarely collision-free, and additional work needs to be done to resolve collisions if they do occur. We will discuss collision resolution techniques in the next section.

※ 17.2.4 *Addressing Methods*

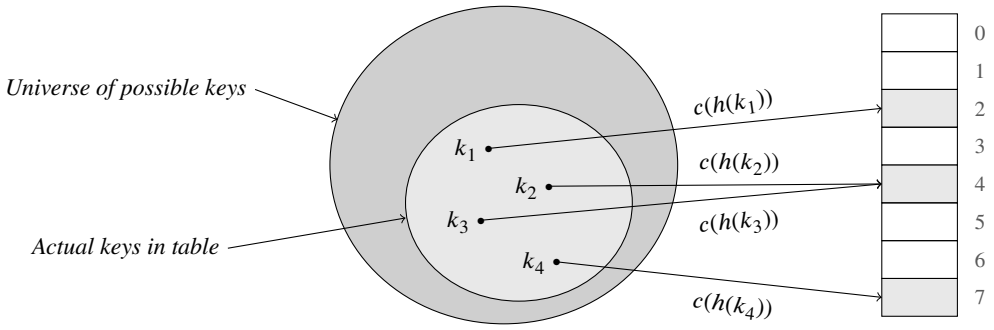
There are two primary methods we can use to search a lookup table when given a key. The first, **direct addressing**, can be used if the universe of possible keys is relatively small. If we know that the possible keys can be drawn from the set $\{0, 1, \ldots, m-1\}$ for some value of m that is not too large, then we can build an array of size m and use the value of the key to directly address this array (hence the name *direct* addressing).



The universe of possible keys can only take on the values of 0-7, so direct addressing can be used with a table of size 8.

However, if the universe of possible keys is so large, or if it greatly exceeds the actual number of keys that are stored in the table, then direct addressing is no longer feasible. In this case, it would be preferable to use **hashed addressing**, where only the actual keys in a hash table are mapped to a position in the underlying array. With hashed addressing, we apply a hash function h (and a compression function c) that maps each key in the universe of possible keys to a value in the set $\{0, 1, \ldots, m-1\}$, where m is the size of the underlying table.

Since the application of hash and compression is not one-to-one, it is inevitable that multiple keys may end up at the same position of the underlying array. As mentioned, this is known as a collision and will need to be resolved using a collision resolution technique.



Hash and compression are applied to a key to determine where to address its value in a table. Because hashing is not one-to-one, collisions may occur (as illustrated with k_2 and k_3 above).

17.3 Collision Resolution Techniques

Let's return to the hash table of prices that we introduced earlier:

Key	`"Apple"`	`"Banana"`	`"Milk"`			`"Flour"`	`"Ginger"`			`"Tofu"`
Value	`3.99`	`2.49`	`2.09`			`2.29`	`2.69`			`3.79`
	0	1	2	3	4	5	6	7	8	9

Suppose we try to add the remaining food item that has not been added yet:

```
{"Avocado", 4.99}
```

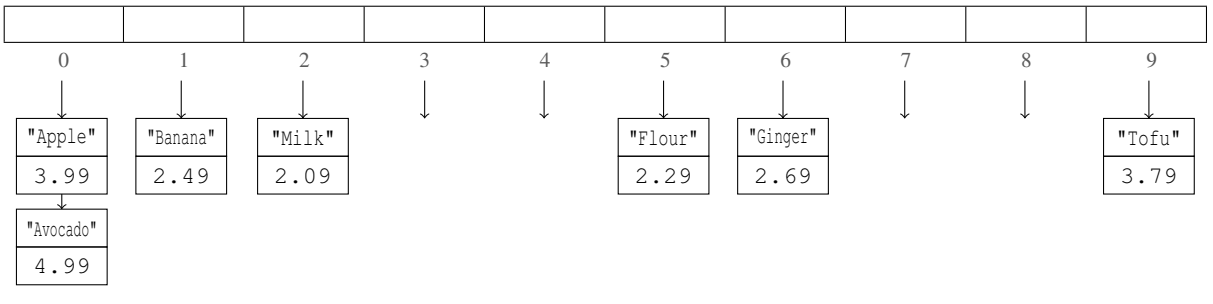
If we were to hash the string `"Avocado"` using our hash function, we would get an index of 0. However, `"Apple"` is already sitting at index 0! In this case, we have a **collision**, where multiple keys hash to the same index. In this section, we will discuss four collision resolution techniques that can be used to resolve collisions:

- separate chaining
- linear probing
- quadratic probing
- double hashing

※ 17.3.1 *Separate Chaining*

The first collision resolution technique we will discuss is **separate chaining**. With separate chaining, each index of the hash table stores a linked list that holds all the keys that hash to that index. That is, index 0 would store a list of all items that hashed to index 0, index 1 would store a list of all items that hashed to index 1, and so on.

If the above hash table used separate chaining, we would resolve the collision by appending `"Avocado"` to the list at index 0. This would produce the following result:



If we had N keys that were evenly distributed across all M indices, the average linked list in our hash table would have a length of N/M. The average-case time complexity of searching for a key would therefore be $\Theta(N/M)$, since it takes constant time to identify which list a key belongs to (via hashing), and there are an average of N/M elements that you have to search once you know the correct list. (In fact, if we have a good distribution and a big enough M, the value of N/M can actually be treated as a constant.)

The complexities of hash table operations using separate chaining are summarized in the table below:

Operation	Average-Case Complexity
Insert Key	$\Theta(1)$ if duplicate keys are allowed, $\Theta(N/M)$ if duplicates not allowed
Search for Key	$\Theta(N/M)$
Erase Key	$\Theta(N/M)$

Insertion takes $\Theta(N/M)$ time on average if duplicate keys are not allowed. This is because you need to check if the key already exists before adding it in. This requires you to iterate through its corresponding list, which has a length of N/M, provided we have a good hash function.

※ 17.3.2 *Linear Probing*

With separate chaining, every element is guaranteed to end up at the index it hashes to. However, this is not always true using the other collision resolution techniques. Linear probing, quadratic probing, and double hashing are all examples of **open addressing** techniques, which use empty locations in the table to resolve collisions. Unlike separate chaining, these methods do not allow an index to store more than a single key-value pair.

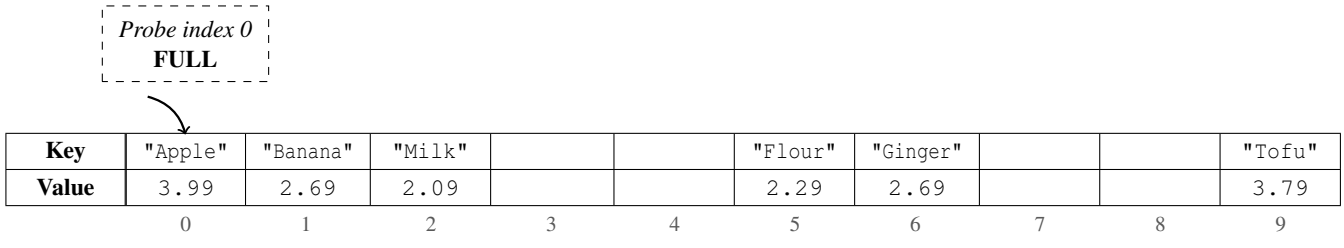
Before we introduce these techniques, we will first introduce the concept of probing. A **probe** is a check that is made to determine whether an index in the table is occupied. For example, if we wanted to insert `"Apple"` into the hash table, we must first probe index 0 of the table to determine if something is already there. There are four possible outcomes of a probe:

- **Empty:** The probe finds an empty cell in the table at an index. An empty cell has never held an item before.
- **Deleted:** The probe finds a cell that once held an item, but is not currently holding an item.
- **Hit:** The probe finds an occupied cell that contains an item whose key matches the search key.
- **Full:** The probe finds an occupied cell, but the key does not match the search key.

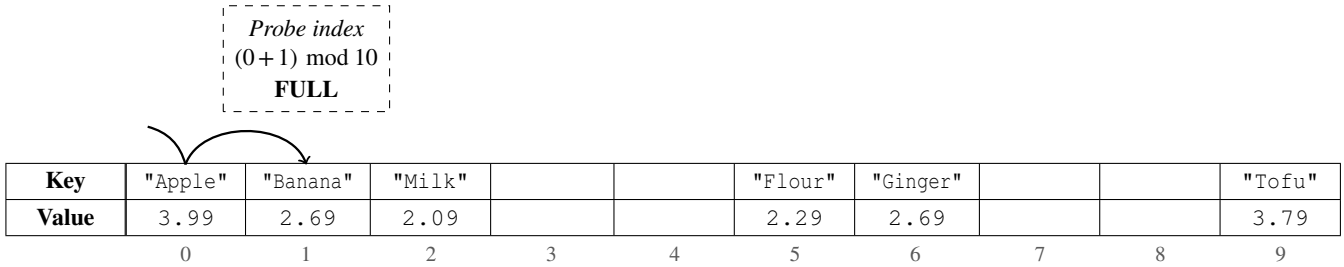
With **linear probing**, we resolve collisions by continuously probing sequential indices until we find an open position (looping back to the beginning if we reach the end). Consider the example table we had before:

	0	1	2	3	4	5	6	7	8	9
Key	`"Apple"`	`"Banana"`	`"Milk"`			`"Flour"`	`"Ginger"`			`"Tofu"`
Value	`3.99`	`2.49`	`2.09`			`2.29`	`2.69`			`3.79`

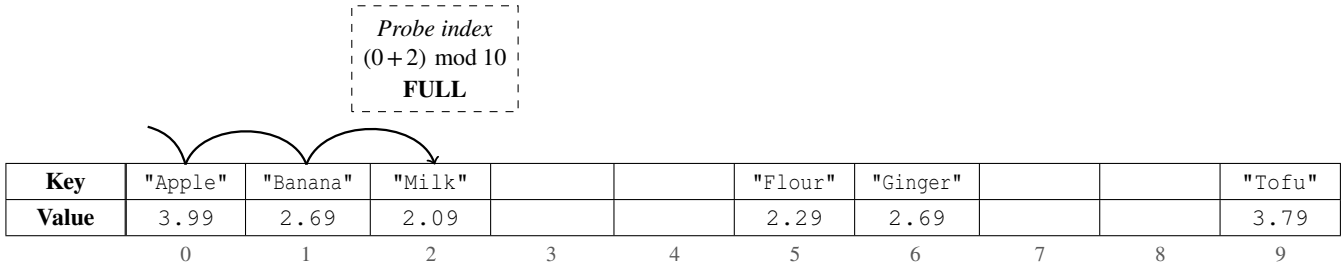
If we attempted to insert `"Avocado"` into this table with linear probing as our collision resolution method, we would first probe index 0 to see if something is already there (as `"Avocado"` hashes to 0). Since `"Apple"` is already there, we would probe sequential indices until we find an open spot — that is, we would probe index $(0 + 1) \bmod M$, then index $(0 + 2) \bmod M$, then index $(0 + 3) \bmod M$, etc. until we find an empty spot. In this case, the first empty cell we probe is at index 3, so `"Avocado"` is placed at index 3. (M is our table size, which is 10.)



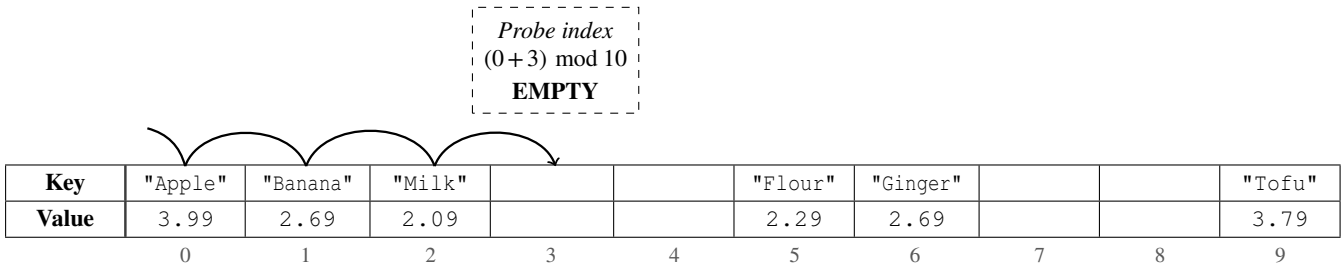
	0	1	2	3	4	5	6	7	8	9
Key	`"Apple"`	`"Banana"`	`"Milk"`			`"Flour"`	`"Ginger"`			`"Tofu"`
Value	`3.99`	`2.69`	`2.09`			`2.29`	`2.69`			`3.79`

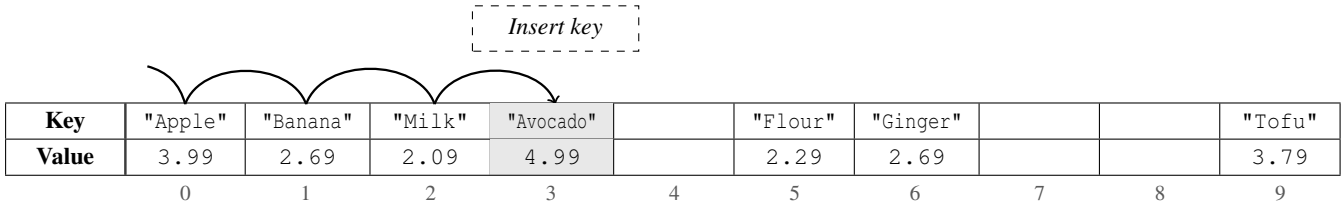


	0	1	2	3	4	5	6	7	8	9
Key	`"Apple"`	`"Banana"`	`"Milk"`			`"Flour"`	`"Ginger"`			`"Tofu"`
Value	`3.99`	`2.69`	`2.09`			`2.29`	`2.69`			`3.79`

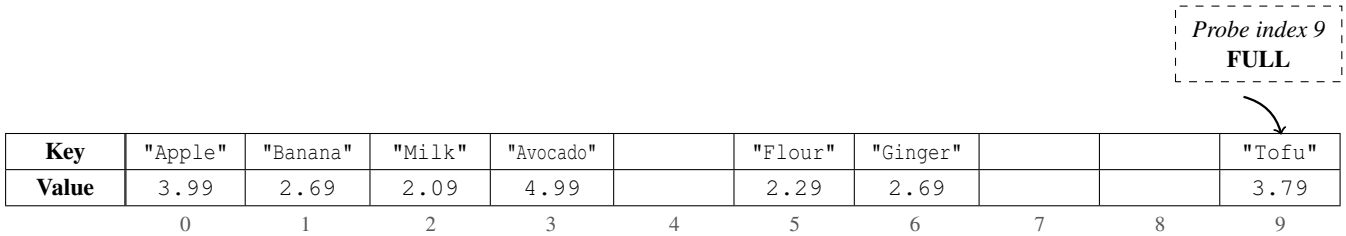


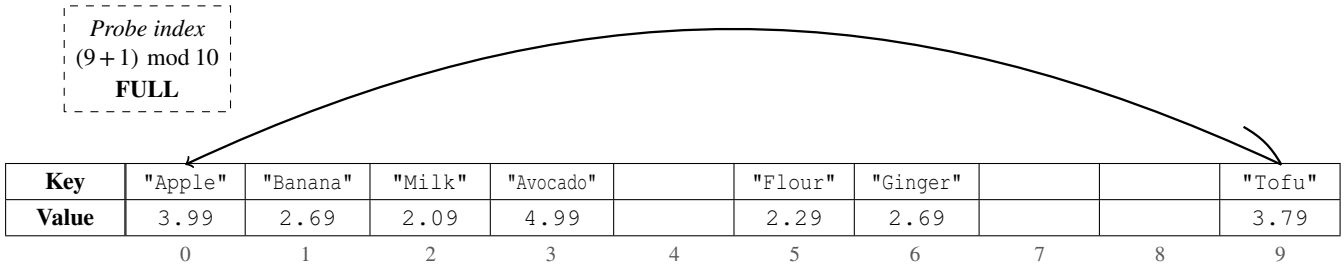
	0	1	2	3	4	5	6	7	8	9
Key	`"Apple"`	`"Banana"`	`"Milk"`			`"Flour"`	`"Ginger"`			`"Tofu"`
Value	`3.99`	`2.69`	`2.09`			`2.29`	`2.69`			`3.79`



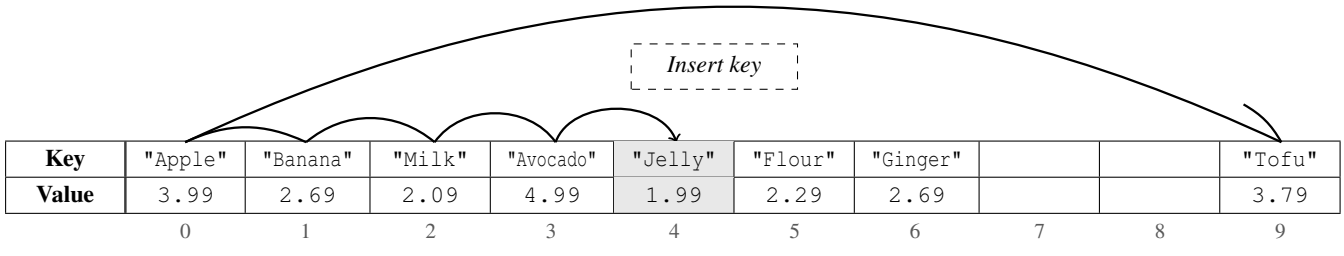


As another example, suppose we wanted to add the key-value pair `{"Jelly", 1.99}` to our hash table. We would first hash `"Jelly"` using our hash function, which returns an index of 9. Then, we would probe index 9 to see if it is available. Since `"Tofu"` already occupies that index, we then check indices $(9 + 1) \bmod M$, $(9 + 2) \bmod M$, etc. The first open index we find is index 4, so `"Jelly"` gets placed at that index.



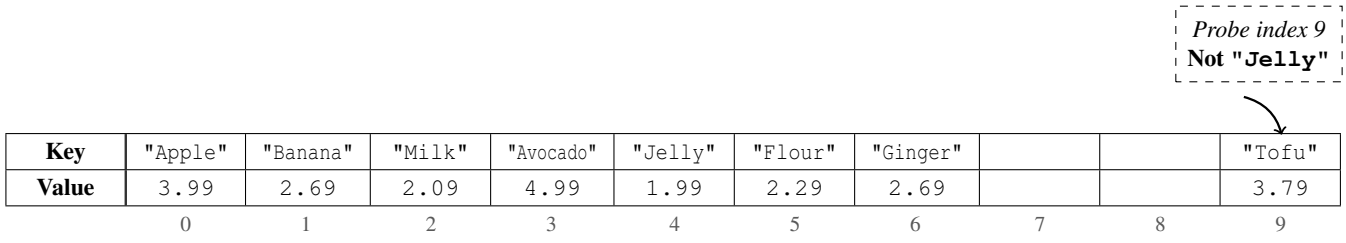


… a few probes omitted …

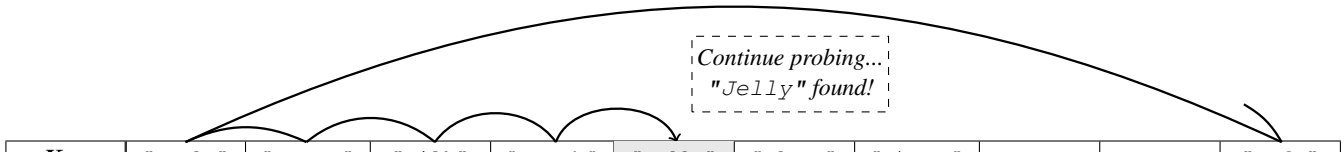


To *search* for a key using linear probing, we first hash the key to find its intended index. Then, we probe that index to determine if the key is there. If it is not, we would continuously probe sequential indices until we either find the key we want, or we find an empty cell (this would mean that the key does not exist in the table).

For instance, suppose we wanted to find the price of `"Jelly"`. We would first input `"Jelly"` into our hash function and obtain an index of 9. We then check index 9 to see if `"Jelly"` is there. It is not, but that does *not* mean that `"Jelly"` does not exist in the table! Since we are using open addressing, it is possible that `"Jelly"` got placed at a different index because of a collision. Thus, we will sequentially probe indices 0, 1, 2, 3, and 4. We find that `"Jelly"` is at index 4, so we return the value at that index (`1.99`).

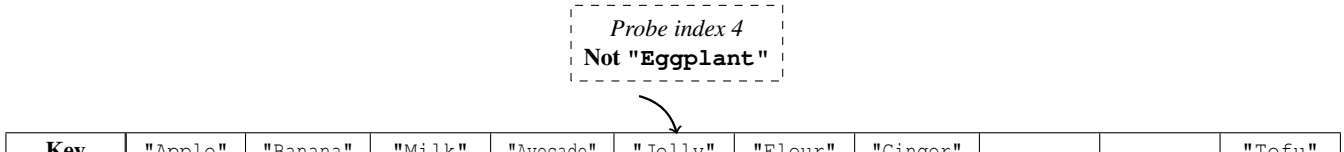


Key	"Apple"	"Banana"	"Milk"	"Avocado"	"Jelly"	"Flour"	"Ginger"			"Tofu"
Value	3.99	2.69	2.09	4.99	1.99	2.29	2.69			3.79
	0	1	2	3	4	5	6	7	8	9

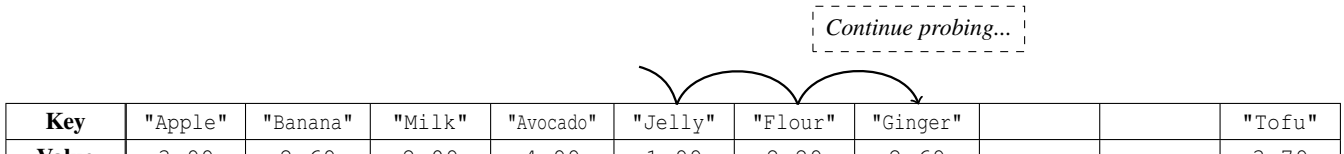


Key	"Apple"	"Banana"	"Milk"	"Avocado"	"Jelly"	"Flour"	"Ginger"			"Tofu"
Value	3.99	2.69	2.09	4.99	1.99	2.29	2.69			3.79
	0	1	2	3	4	5	6	7	8	9

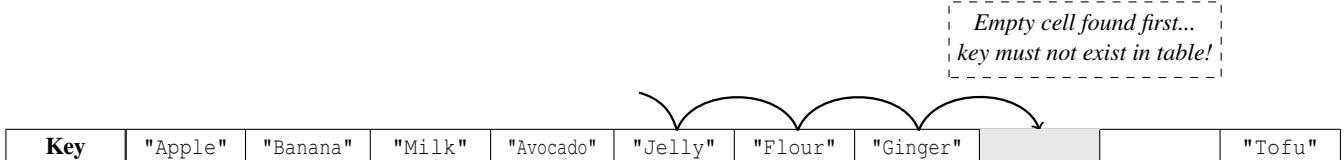
On the other hand, suppose we wanted to find the price of a non-existent item like `"Eggplant"`. Passing `"Eggplant"` into the hash table gives us an index of 4, but `"Jelly"` occupies that index. We would then probe indices 5, 6, and 7 to make sure that `"Eggplant"` was not placed at a different index due to a collision. The keys in indices 5 and 6 do not match, but index 7 is empty. *Once we probe an empty cell* (note that the cell must be *empty* and not *deleted* — the distinction is important), *we can immediately stop probing.* This is because an empty cell has never held an item before (by definition), so any key we are searching for would not have made it past a probe at this empty position when it was first inserted into the table (since we stop probing as soon as we find an empty spot). In this example, `"Eggplant"` cannot exist in the table since it would have been placed at index 4, 5, 6, or 7 if it did exist.



Key	"Apple"	"Banana"	"Milk"	"Avocado"	"Jelly"	"Flour"	"Ginger"			"Tofu"
Value	3.99	2.69	2.09	4.99	1.99	2.29	2.69			3.79
	0	1	2	3	4	5	6	7	8	9



Key	"Apple"	"Banana"	"Milk"	"Avocado"	"Jelly"	"Flour"	"Ginger"			"Tofu"
Value	3.99	2.69	2.09	4.99	1.99	2.29	2.69			3.79
	0	1	2	3	4	5	6	7	8	9



Key	"Apple"	"Banana"	"Milk"	"Avocado"	"Jelly"	"Flour"	"Ginger"			"Tofu"
Value	3.99	2.69	2.09	4.99	1.99	2.29	2.69			3.79
	0	1	2	3	4	5	6	7	8	9

Removing elements from the table is slightly trickier. Suppose we wanted to erase the key `"Tofu"` from the table. We use our hash function to convert `"Tofu"` to the index 9, and then we go to index 9 to delete the value (after checking that the keys match). After the deletion, the table would look like this:

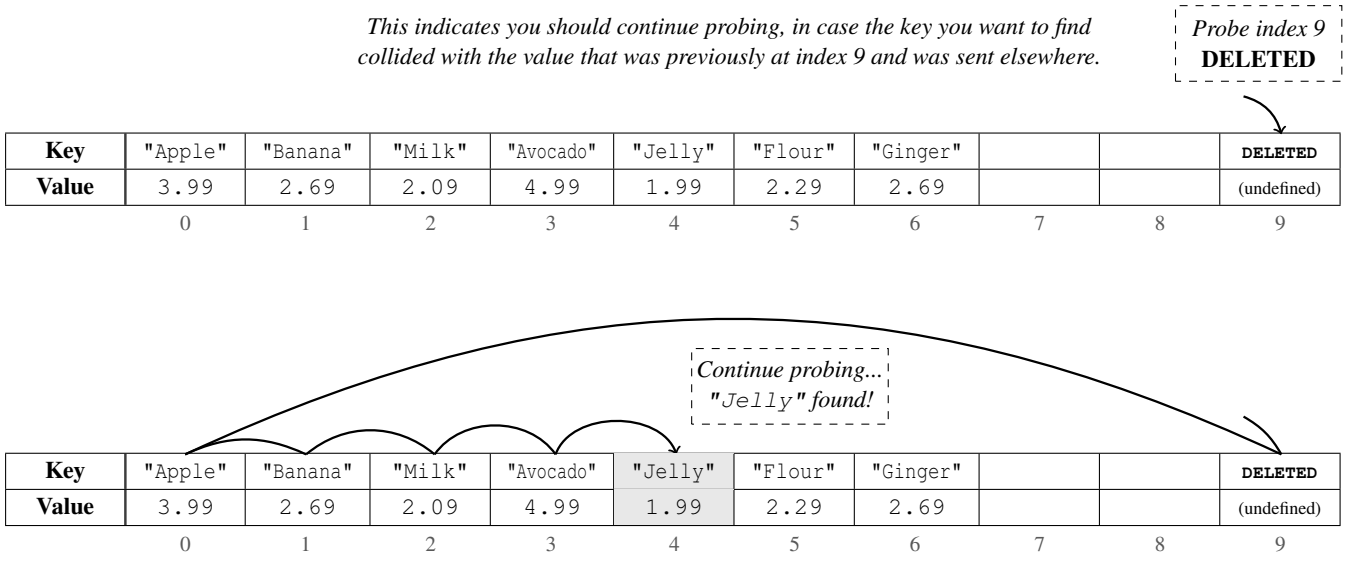
Key	`"Apple"`	`"Banana"`	`"Milk"`	`"Avocado"`	`"Jelly"`	`"Flour"`	`"Ginger"`			
Value	`3.99`	`2.69`	`2.09`	`4.99`	`1.99`	`2.29`	`2.69`			
	0	1	2	3	4	5	6	7	8	9

However, this is actually an invalid table! Why? Suppose we wanted to retrieve the price of `"Jelly"` again. We convert `"Jelly"` to the index 9, but see that index 9 is empty. As a result, we would incorrectly conclude that `"Jelly"` does not exist in the table! This ended up happening because `"Jelly"` was added to the table after `"Tofu"`, and the presence of `"Tofu"` caused `"Jelly"` to be placed at a different index. By removing `"Tofu"`, we ended up losing access to all elements that had previously collided with `"Tofu"` when they were inserted!

To address this, we will need to add a special flag whenever we delete an element. This flag lets the hash table know that a specific index previously held an element that is no longer there, and that elements that had previously collided with this deleted element may be elsewhere in the table. Using the above example of deleting `"Tofu"` and adding a special flag, the table would look like this:

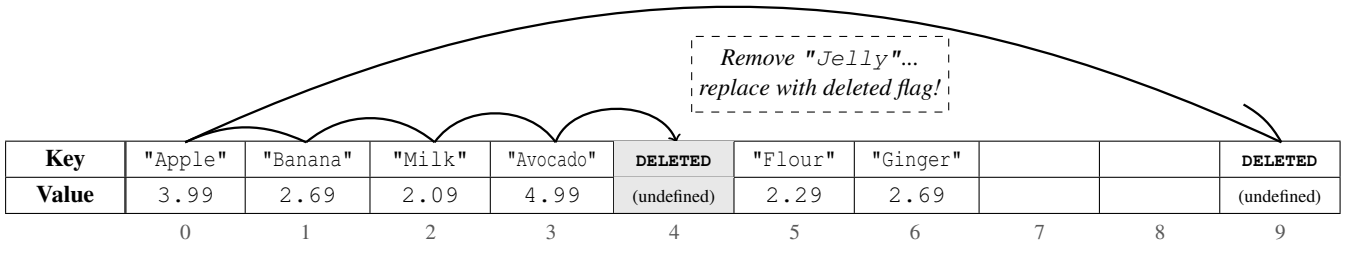
Key	`"Apple"`	`"Banana"`	`"Milk"`	`"Avocado"`	`"Jelly"`	`"Flour"`	`"Ginger"`			**DELETED**
Value	`3.99`	`2.69`	`2.09`	`4.99`	`1.99`	`2.29`	`2.69`			(undefined)
	0	1	2	3	4	5	6	7	8	9

Now, if we want to retrieve the price of `"Jelly"`, we would see that index 9 has a "deleted" element, and that we should continue probing for `"Jelly"` because it could have been sent elsewhere after a collision with the deleted element. We then probe indices 0, 1, 2, 3, and 4 before we find `"Jelly"` (and return its price).



Key	`"Apple"`	`"Banana"`	`"Milk"`	`"Avocado"`	`"Jelly"`	`"Flour"`	`"Ginger"`			**DELETED**
Value	`3.99`	`2.69`	`2.09`	`4.99`	`1.99`	`2.29`	`2.69`			(undefined)
	0	1	2	3	4	5	6	7	8	9

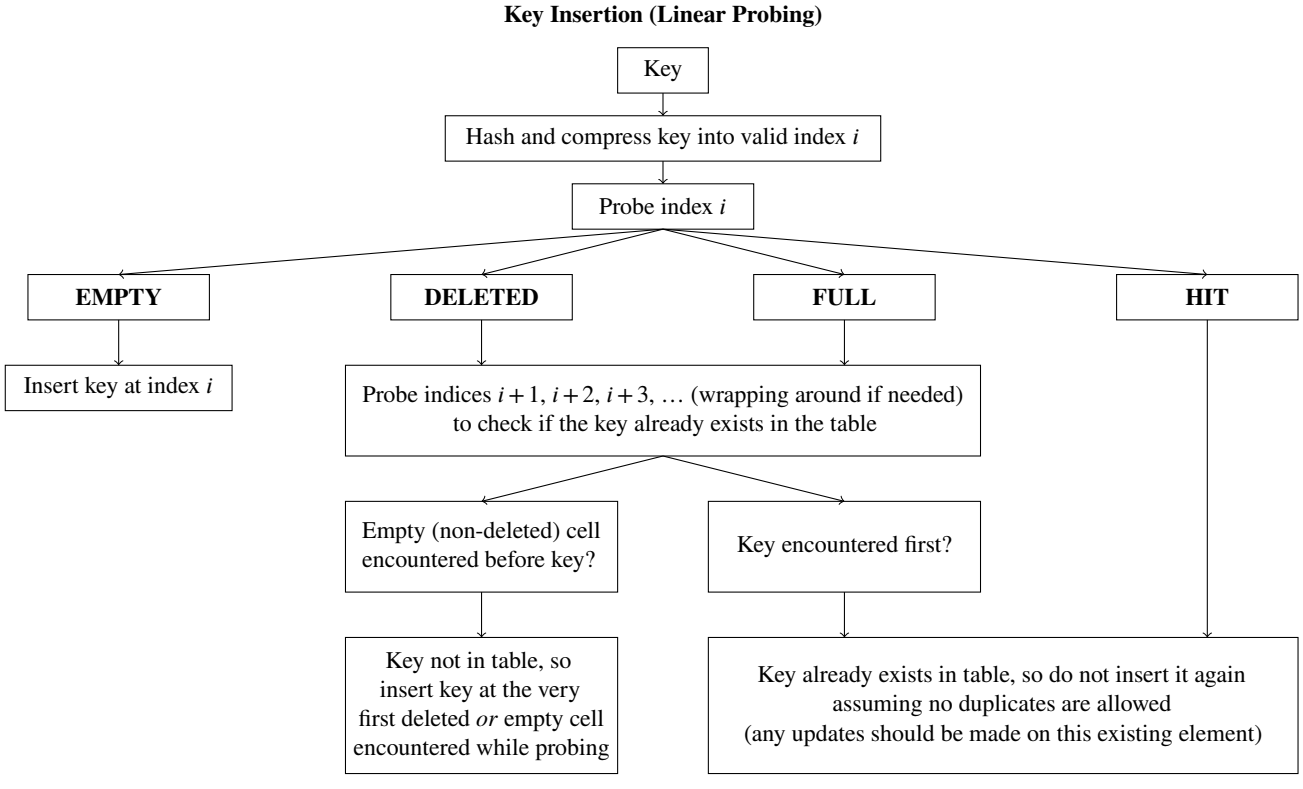
If we wanted to delete `"Jelly"` from the table, we would follow the same process as before to identify the position of this key, and then we would replace it with the deleted flag so that future queries would know an element had existed at this position previously.

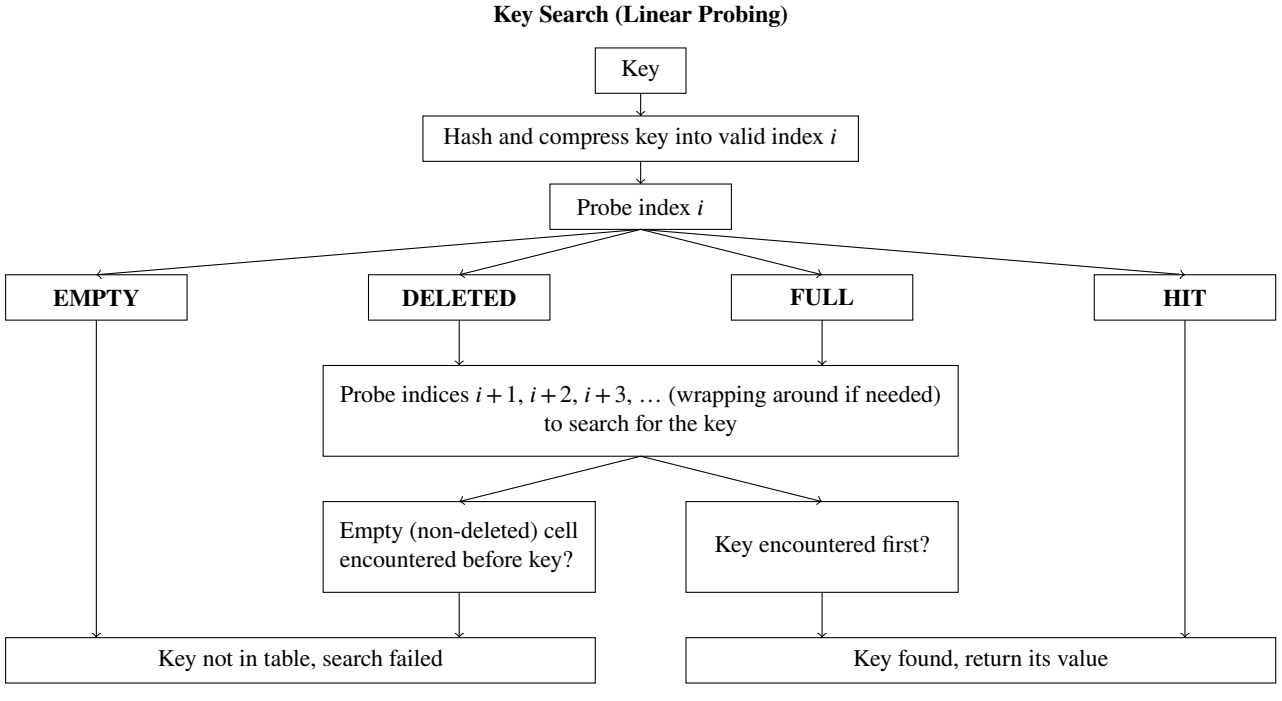


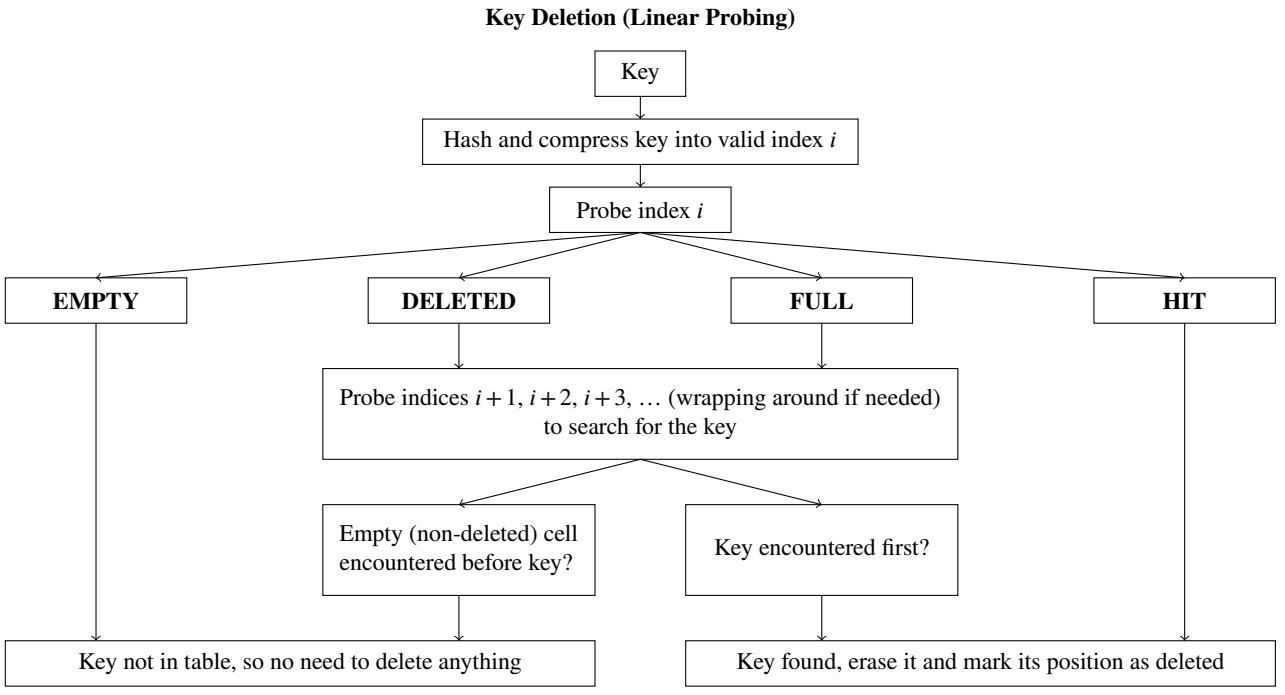
Key	`"Apple"`	`"Banana"`	`"Milk"`	`"Avocado"`	**DELETED**	`"Flour"`	`"Ginger"`			**DELETED**
Value	`3.99`	`2.69`	`2.09`	`4.99`	(undefined)	`2.29`	`2.69`			(undefined)
	0	1	2	3	4	5	6	7	8	9

Deleted elements add a twist to how we conduct insertions and searches. When trying to search for an element, we treat a "deleted" position as if it were occupied and keep on looking. For insertion (assuming that duplicate keys are not allowed), we treat a "deleted" position as occupied while looking to see if the element already exists, but as an empty spot during the actual insertion process. For example, if we wanted to re-add `"Tofu"` to the hash table, we would first have to probe indices 9, 0, 1, 2, 3, 4, 5, 6, and 7 to confirm that `"Tofu"` does not already exist in the table. However, when we actually insert `"Tofu"` into the table, it should be written to index 9 (the first *deleted* cell we encountered) and not index 7 (the first *empty* cell we encountered), as positions that are deleted do not hold valid items and can be inserted into.

The following diagrams provide a summary of insertion, search, and deletion on a hash table that uses linear probing to resolve collisions:







Example 17.1 Consider a hash table of size 10 and a hash function $h(k) = k$, where collisions are handled using linear probing. If $h(k)$ exceeds the table size of 10, it is compressed down to a valid index by taking its modulo with 10, i.e., $c(k) = h(k) \bmod 10$. After the following operations, what does the table look like? Duplicate keys are not allowed, and the table does not get resized.

1. Insert 13
2. Insert 23
3. Insert 33
4. Insert 14
5. Erase 23
6. Insert 14
7. Erase 13
8. Insert 43

First, let's insert 13. After compression, 13 maps to 13 mod 10 = 3, so it gets placed at index 3.

			13						
0	1	2	3	4	5	6	7	8	9

Next, let's insert 23. Like 13, 23 also maps to 23 mod 10 = 3. However, since 13 is already at index 3, we have a collision. Since we are using linear probing, we would put 23 at the next available position, or index 4.

			13	23					
0	1	2	3	4	5	6	7	8	9

33 also maps to 3, so we place it at index 5 — the next available position — using linear probing.

			13	23	33				
0	1	2	3	4	5	6	7	8	9

Now, let's insert 14. 14 maps to 14 mod 10 = 4, but there is already another value at index 4. With linear probing, we find the next open position (index 6) and put 14 there.

			13	23	33	14			
0	1	2	3	4	5	6	7	8	9

Next, we erase 23. Since 23's mapped index after compression is 3, we expect it to be at index 3. However, since 13 is actually at index 3, we have to probe an additional index to discover that 23 is at index 4. We cannot erase 23 completely — as mentioned earlier, we will need to mark index 4 with a "deleted" flag for future operations.

			13	DELETED	33	14			
0	1	2	3	4	5	6	7	8	9

Next, we will insert 14. 14 maps to 14 mod 10 = 4, so we would write 14 to index 4. However, even though index 4 does not currently store anything, it is marked as "deleted". Thus, we will have to continue probing since 14 might already exist in the table, but in a different position because of a collision with this deleted element. After probing, we notice that 14 is already at index 6, so no insertion is actually made.

			13	DELETED	33	14			
0	1	2	3	4	5	6	7	8	9

Next, we erase 13. Like before, we mark the location of 13 (index 3) with a "deleted" flag.

			DELETED	DELETED	33	14			
0	1	2	3	4	5	6	7	8	9

Next, we insert 43. 43 maps to 43 mod 10 = 3, so we look at index 3. Index 3 does not store a valid element, but it is marked with a "deleted" flag. Thus, we need to continue probing to determine if 43 is already in the table. The next index we probe is index 4, which is also "deleted", so we continue on. Index 5 and index 6 do not store 43, and index 7 is empty. This empty cell indicates that 43 does not exist in the table. 43 is then inserted at the first deleted or empty cell encountered during the probing process, which is index 3. It is okay to write to index 3 because "deleted" cells do not store actual data — the "deleted" flag is simply a marker to let the algorithm know to continue probing during a search.

			43	DELETED	33	14			
0	1	2	3	4	5	6	7	8	9

This is the state of the hash table after all eight operations are completed.

Example 17.2 Consider the following hash table in the previous example (which uses linear probing):

			43	DELETED	33	14			
0	1	2	3	4	5	6	7	8	9

How many buckets need to be probed before you can safely insert the key 53 into this table?

Before we can insert 53 into the table, we must first make sure that it is not in the table already. In this case, 53 hashes to 53 mod 10 = 3, so we first probe index 3. Since 43 is at that spot, we then probe index 4. Index 4 contains a "deleted" item, which lets us know that we should continue probing. We then probe index 5, which is not equal to 53. We then probe index 6, which also is not equal to 53. We then probe index 7, which is empty. Since we reached an empty bucket without encountering 53, we know that 53 is not in the table, and that we can safely insert it in. The total number of buckets we probed before inserting 53 is therefore 5: indices 3, 4, 5, 6, and 7. Note that 53 would be inserted at index 4, since that is the first non-occupied index we encountered while probing.

			43	53	33	14			
0	1	2	3	4	5	6	7	8	9

※ 17.3.3 *Quadratic Probing*

One disadvantage of linear probing is that it is susceptible to **clustering**. Clustering happens when a group of adjacent indices in a hash table are all occupied. Because open spots are discovered and filled sequentially during a collision with linear probing, there is a tendency for many keys to end up right next to each other (this is specifically known as *primary clustering*). This is not ideal, since the number of collisions would grow as more elements are added.

To see why this is the case, consider two different hash tables that are each half full (i.e., $N = M/2$). In one hash table, the elements are evenly distributed (alternating empty and occupied cells). In the other hash table, all the elements are clustered together. These two hash tables are shown below, where an X represents an occupied cell:

X		X		X		X	⋯		X		X		X	

X	X	X	X	X	X	X	⋯							

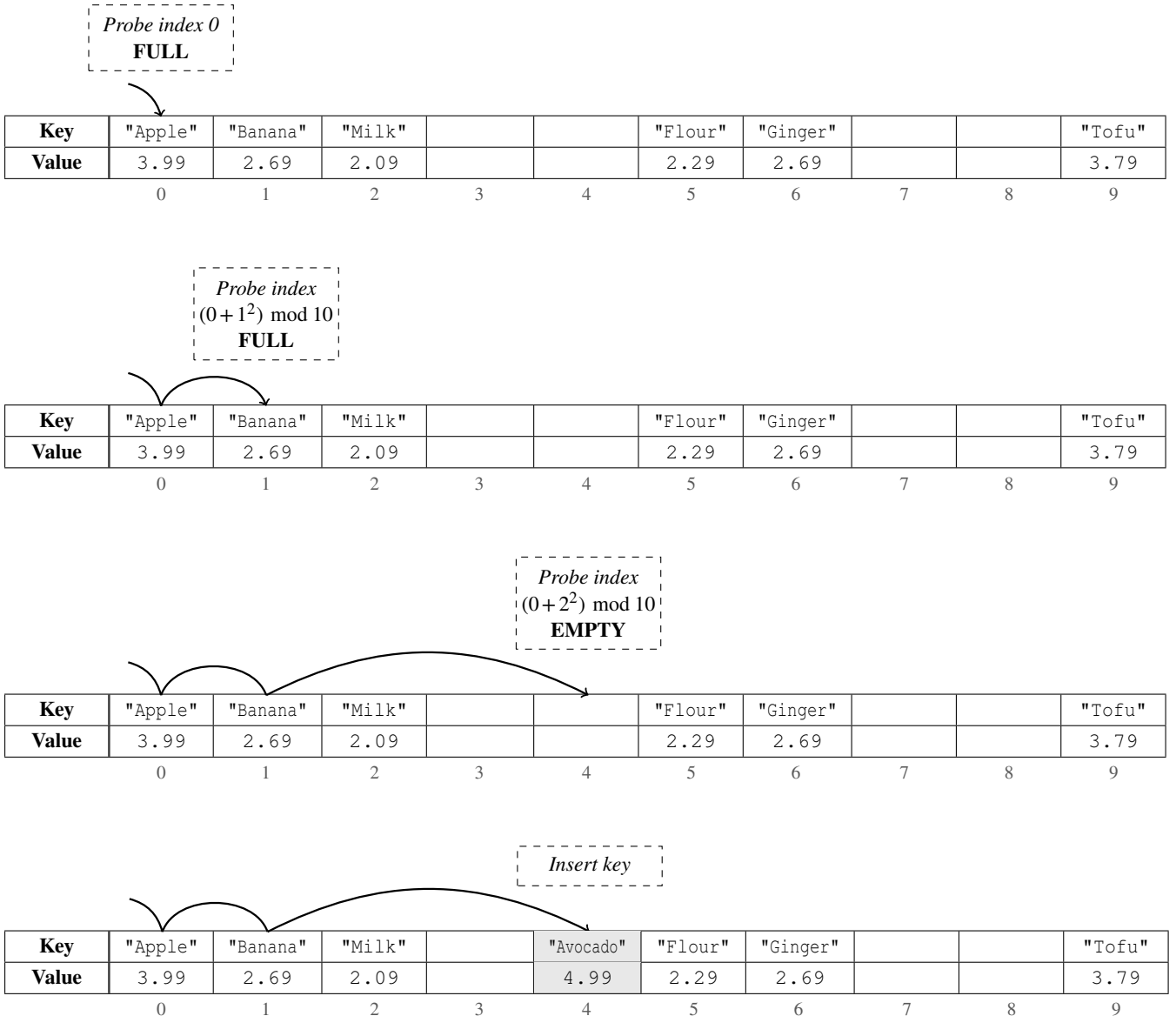
What is the average cost of a hash table operation if the elements are evenly distributed (the first hash table)? Since the hash table is half filled (and assuming that our hash function is good), there is a 50% chance that a key will be hashed to an empty cell. When this happens, we only need to make a single probe. The other 50% of the time, the key will be hashed to an occupied cell. However, since every occupied cell is followed by an empty one, we only need to make two probes in this situation (one for the occupied cell, one for the empty cell next to it). The average cost of a hash table operation is thus $0.5 \times 1 + 0.5 \times 2 = 1.5$ for the first hash table. This is $\Theta(1)$.

What about the second hash table? There is still a 50% chance that a key will be hashed to an empty cell, which will only require one probe. However, there is also a 50% chance a key hashes to a position within the cluster. When this happens, we will have to continuously probe until we reach the end of the cluster. The exact number of probes we need to make depends on where we land (we only need to make two probes if a key gets hashed to the end of the cluster, but we would need to make approximately n probes if a key gets hashed to the beginning of the cluster). However, on average, you would need to make $n/2$ probes, since that is the average distance from an index within the cluster to the end of the cluster. We can thus compute the average cost of a hash table operation to be $0.5 \times 1 + 0.5 \times n/2$, which is $\Theta(n)$. This is much worse than the $\Theta(1)$ time we obtained when no clustering was present!

How can we minimize the likelihood of this happening? One possibility would be to switch from linear probing to **quadratic probing** when handling collisions. Quadratic probing is essentially the same as linear probing, but with one key difference: rather than probing indices $i+1$, $i+2$, $i+3$, …, when a collision occurs at index i, quadratic probing probes indices $i+1^2$, $i+2^2$, $i+3^2$, …, instead. By squaring the distance with every collision, we reduce the risk of clustering, which could bring down the efficiency of our hash table. Let's consider the following table:

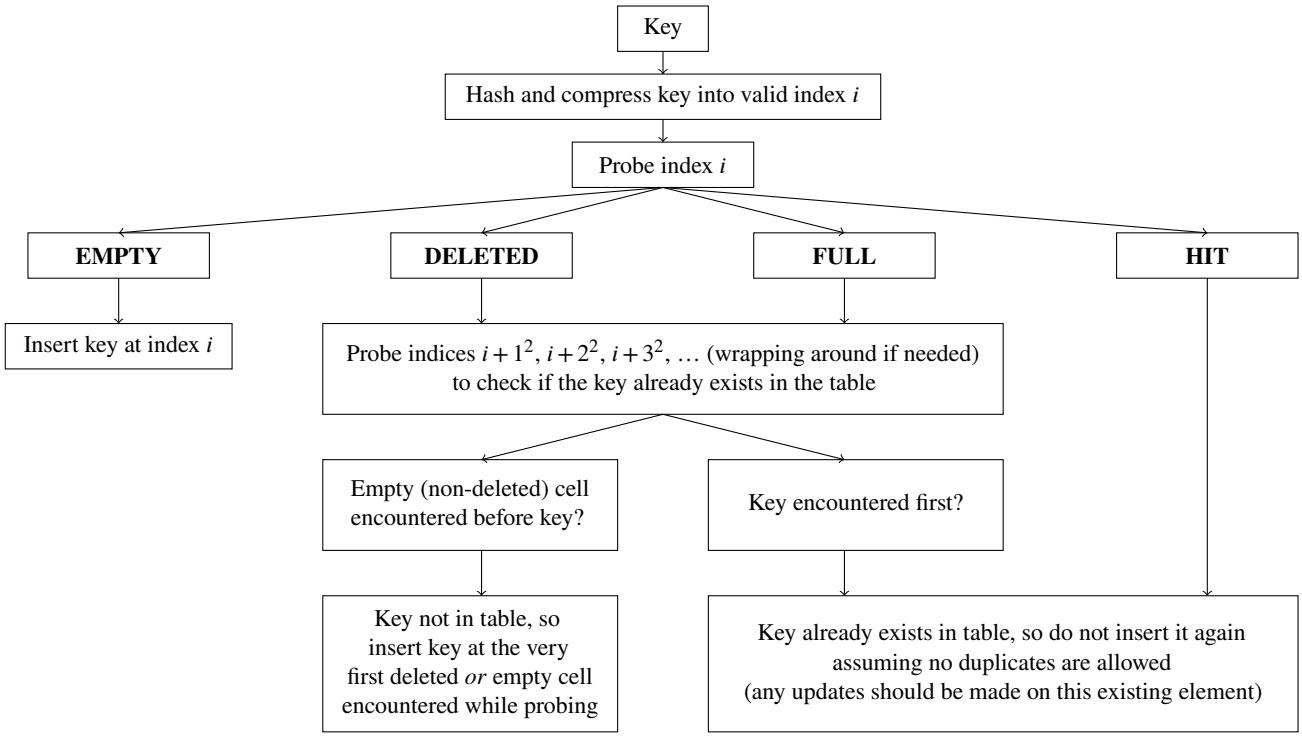
Key	`"Apple"`	`"Banana"`	`"Milk"`			`"Flour"`	`"Ginger"`			`"Tofu"`
Value	`3.99`	`2.49`	`2.09`			`2.29`	`2.69`			`3.79`
	0	1	2	3	4	5	6	7	8	9

Again, let's attempt to insert `"Avocado"` into this hash table, but with quadratic probing as our collision resolution method. `"Avocado"` hashes to index 0, but `"Apple"` is already at index 0. Since index 0 is taken, we check to see if index $0 + 1^2 = 1$ is available. It is not. We then check to see if index $0 + 2^2 = 4$ is available. It is, so `"Avocado"` gets added to index 4.

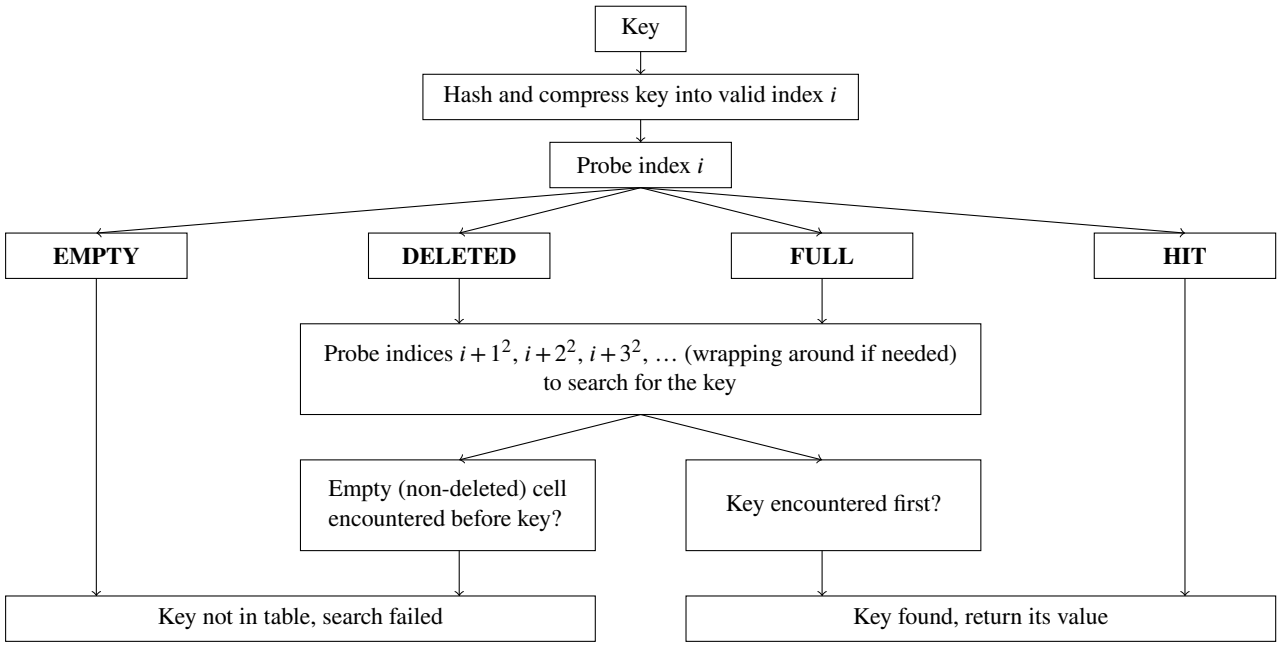


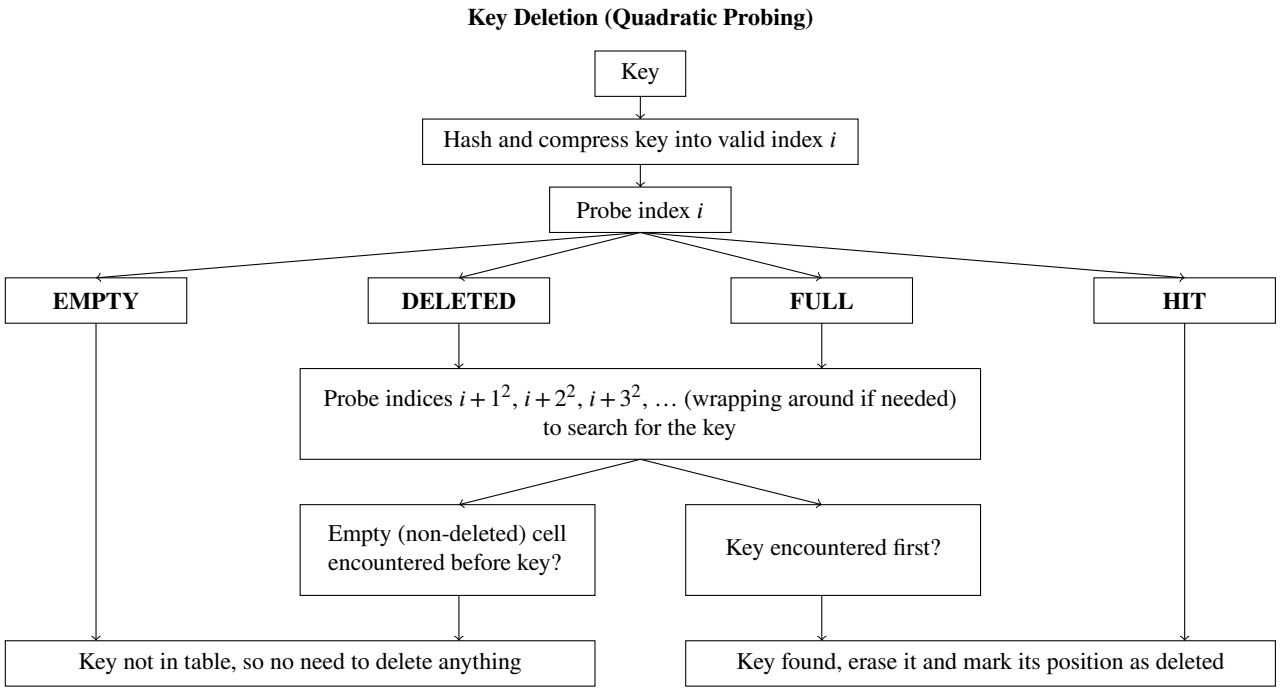
The rules for quadratic probing are shown on the next page. They are nearly identical to the rules for linear probing; the only difference is the sequence of positions probed upon collision.

Key Insertion (Quadratic Probing)



Key Search (Quadratic Probing)





Example 17.3 Consider a hash table with 10 buckets and a hash function $h(k) = k$, where collisions are handled using quadratic probing. If $h(k)$ exceeds the table size of 10, it is compressed down to a valid index by taking its modulo with 10, i.e., $c(k) = h(k) \bmod 10$. After inserting the keys 13, 23, 33, 43, and 53 into this hash table, what does the table look like?

After compression, 13 maps to index 3, so it gets placed at index 3.

			13						
0	1	2	3	4	5	6	7	8	9

23 also maps to index 3, but index 3 is taken. Thus, we check if index $3 + 1^2 = 4$ is taken. It is not, so we place 23 at index 4.

			13	23					
0	1	2	3	4	5	6	7	8	9

33 also maps to index 3, but index 3 is taken. We then check index $3 + 1^2 = 4$, but that is taken as well. We then check index $3 + 2^2 = 7$, which is available, so 33 gets placed at index 7.

			13	23			33		
0	1	2	3	4	5	6	7	8	9

43 also maps to index 3, but index 2 is taken. We then check indices $3 + 1^2$ and $3 + 2^2$, but both are taken as well. We then check $3 + 3^2 = 12$ (which wraps around to $12 \bmod 10 = 2$). Index 2 hasn't been taken yet, so 43 gets placed at index 2.

		43	13	23			33		
0	1	2	3	4	5	6	7	8	9

Lastly, 53 also maps to index 3. Index 3 is taken, so we check $3 + 1^2$, $3 + 2^2$, $3 + 3^2$, $3 + 4^2$ for an available position. Everything before $3 + 4^2$ is taken. However, the hash value $3 + 4^2$ wraps around to index $19 \bmod 10 = 9$, and index 9 is empty. Thus, 53 gets placed at index 9.

		43	13	23			33		53
0	1	2	3	4	5	6	7	8	9

This is the state of the hash table after all five keys are inserted using quadratic probing.

※ 17.3.4 *Double Hashing*

Compared to linear probing, quadratic probing is better at preventing primary clustering from occurring in a hash table. However, even if quadratic probing is used, elements that hash the same position will still always have the same probe sequence, regardless of how far they land from their hashed position (which results in a phenomenon known as *secondary clustering*). For example, if you tried to insert 63, 73, 83, …, into the hash table of the previous example, you will always probe index 3, then 4, then 7, then 9, and so on. To solve this problem, we can use a collision resolution technique known as **double hashing**. With double hashing, we apply an additional hash function to the key if a collision occurs to determine which cells we should subsequently probe for open spots. That way, if a collision occurs, we might end up checking index $i+3$ for one key, index $i+5$ for a different key, index $i+8$ for another key, etc. instead of checking the same sequence of $i+1$, $i+4$, $i+9$, etc. every time. The double hashing formula is shown below:

$$t(key)+j\times t'(key)$$

Here, $t'(key)$ represents a second hash that is used if there is a collision, and j represents the collision number.

Let's break this formula down. If double hashing is used (with modulus compression), we first probe index $[t(key) \bmod M]$ to check if it is empty. If it is, we add the key to that index location. However, if we have a collision, the next index we check is

$$[t(key)+1\times t'(key)] \bmod M$$

If there is a collision at this index, the next index we would check is

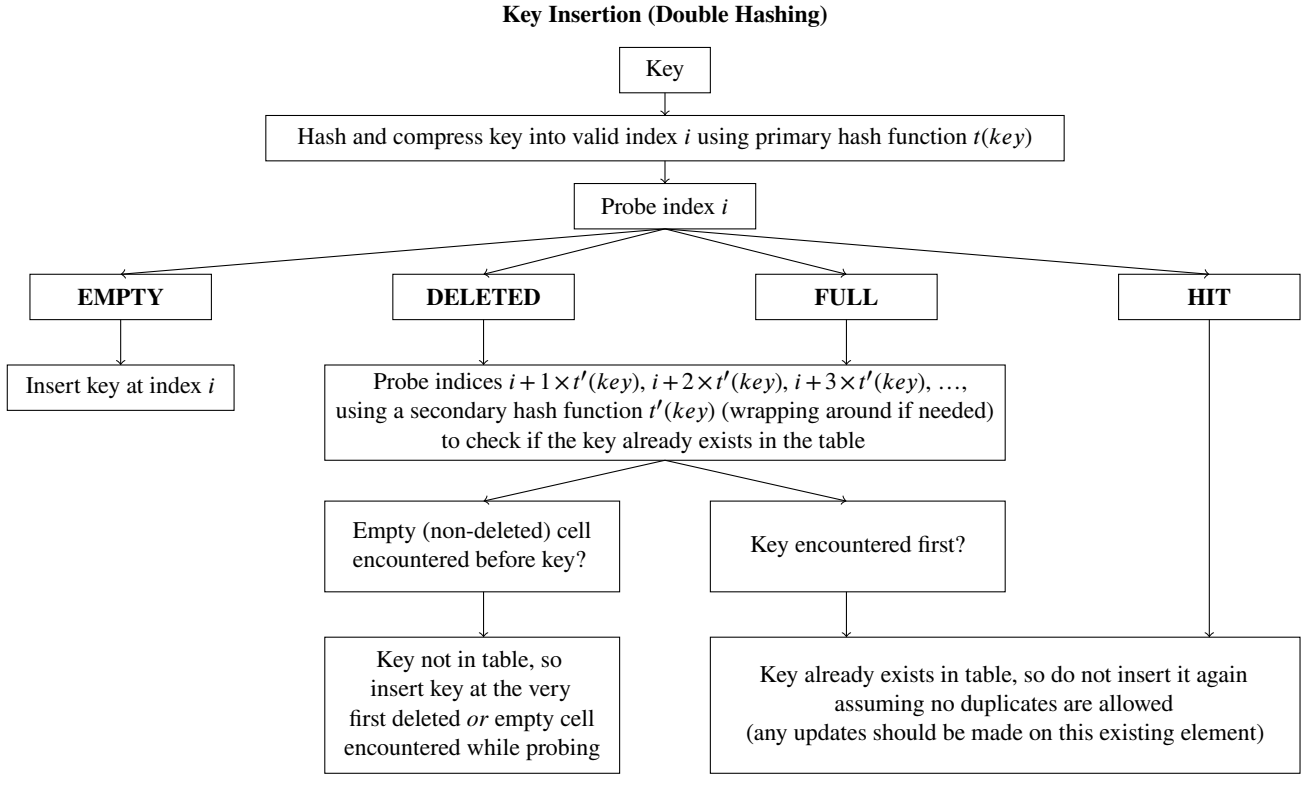
$$[t(key)+2\times t'(key)] \bmod M$$

If there is still a collision, the next index we would check is

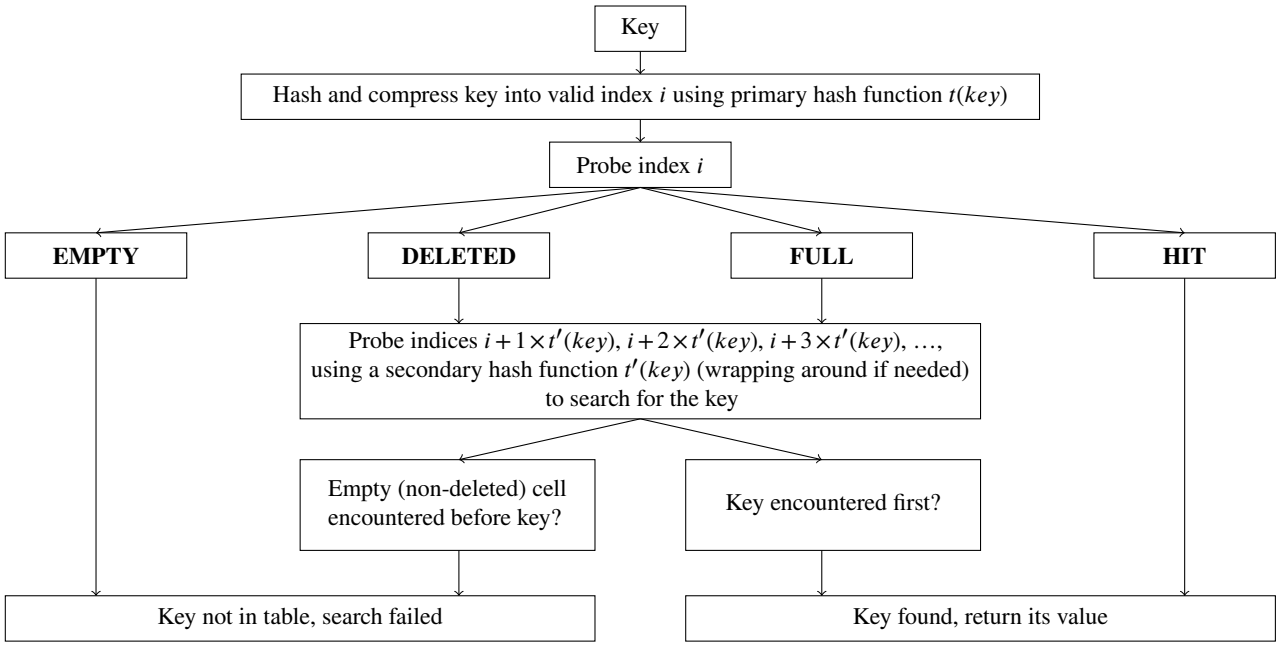
$$[t(key)+3\times t'(key)] \bmod M$$

We would continue incrementing the collision number until we get an index that is not occupied.

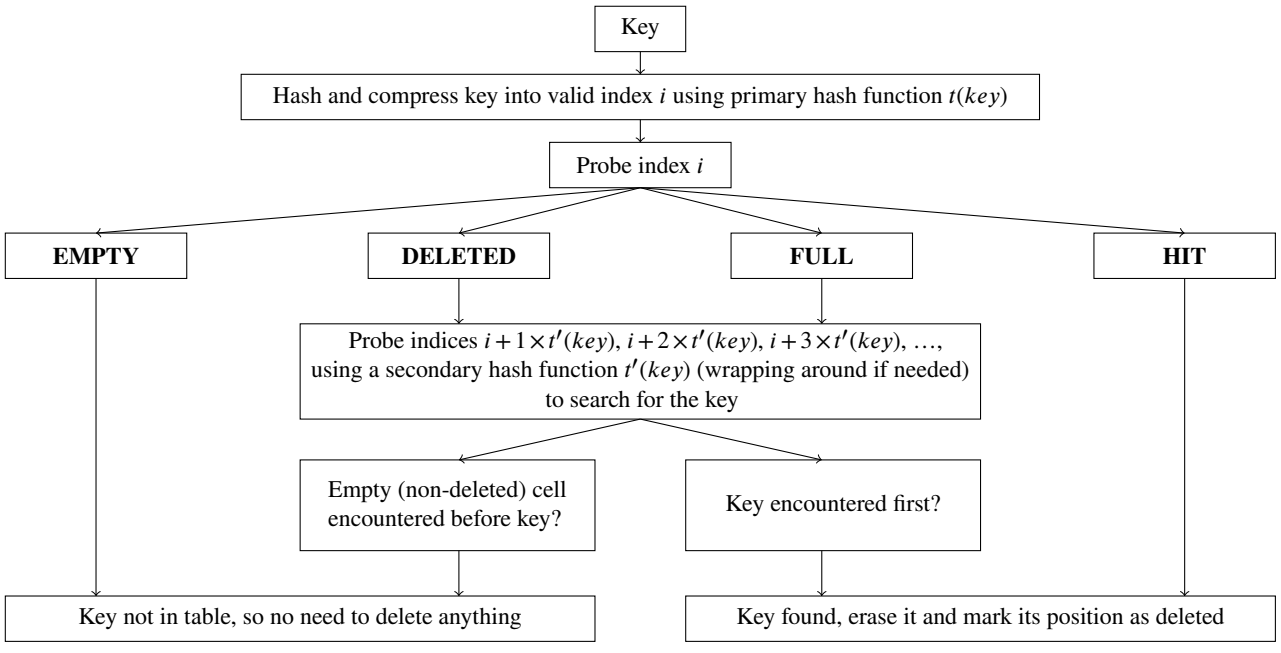
To summarize, in double hashing, you are given a primary hash function $t(key)$ and a secondary hash function $t'(key)$. The primary hash function is applied to the key to determine the index it should go to, and the secondary hash function is applied to determine the locations to probe during a collision. Instead of probing indices $(i+1) \bmod M$, $(i+4) \bmod M$, $(i+9) \bmod M$, …, we probe indices $[i+1\times t'(key)] \bmod M$, $[i+2\times t'(key)] \bmod M$, $[i+3\times t'(key)] \bmod M$, …, and so on. For different keys, the value of $t'(key)$ may be different; this allows us to diversify the probing sequence for each key, which thereby diminishes the occurrence of secondary clustering.



Key Search (Double Hashing)



Key Deletion (Double Hashing)

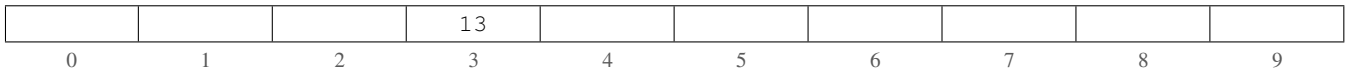


Example 17.4 Consider a hash table with 10 buckets and a hash function $h(k) = k$, where collisions are handled using double hashing. The double hashing formula used is

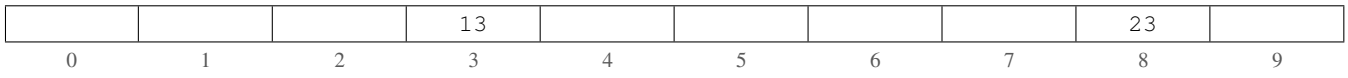
$$t(key) + j(7 - (t(key) \bmod 7))$$

where $t(key)$ represents the integer value the key normally hashes to and j is the collision number. The secondary hash function is $t'(key) = 7 - (t(key) \bmod 7)$. The compression function takes the modulo of $h(k)$ with the table size 10 to return a valid index. After the keys 13, 23, 33, and 25 are inserted into the table, in this order, what does the table look like?

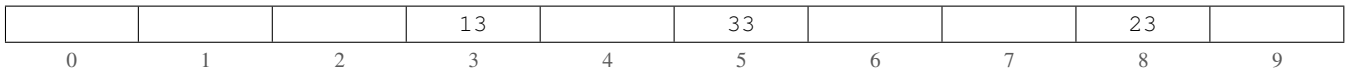
We first pass 13 into our primary hash function, which returns an index of 3 after compression. Since nothing is at index 3, 13 is placed there.



Next, 23 also hashes to index 3 using our primary hash function. Since 13 is already there, we have a collision. Using double hashing, the next index we probe is [23 + 1 × (7 - (23 mod 7))] mod 10 = 28 mod 10 = 8. Index 8 is empty, so 23 gets placed at index 8.



Next, 33 also maps to index 3 using our primary hash function. Since 13 is there, we then check index [33 + 1 × (7 - (33 mod 7))] mod 10 = 35 mod 10 = 5. Index 5 is empty, so 33 gets placed at index 5.



Next, 25 maps to index 5 using our primary hash function. Since 33 is already there, we then check index [25 + 1 × (7 - (25 mod 7))] mod 10 = 28 mod 10 = 8. We end up getting another collision, so we increment our collision number and check index [25 + 2 × (7 - (25 mod 7))] mod 10 = 31 mod 10 = 1. Index 1 is empty, so 25 gets placed at index 1.

This is the state of the hash table after all four keys are inserted.

17.4 Load Factor and Dynamic Hashing

※ 17.4.1 *Load Factor*

The collision resolution technique we use is just one factor that plays a role in the efficiency of a hash table. Another factor is the *table size*. The more full a table becomes, the more collisions you will get, and the less efficient your hash table will be. As a result, a good hash table will need to be dynamically resized based on the size of the data.

Before we discuss table resizing, we will introduce a concept known as the **load factor**, denoted by α. The value of α is N/M, where N represents the number of keys in the table, and M represents the size of the underlying table. We have seen N/M before when discussing separate chaining: with a good hash function, α represents the *average number of items in each list*. However, for open addressing techniques, α takes on a different meaning — it represents the *percentage of table indices that are filled*. Unlike separate chaining, each index in open addressing can hold at most one element, so α must be less than or equal to 1 if open addressing is used (since you cannot have more elements than indices). Note that deleted elements do not contribute to N (since "deleted" is just a bookkeeping flag, not an actual key in the table).

In 1962, Don Knuth proved a set of equations on the expectation of linear probing. If linear probing is used and the hash function distributes keys evenly, the expected number of probes required to successfully search for an existing element in a hash table with load factor α is

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

On the other hand, the expected number of probes required to either (1) unsuccessfully search for a non-existent element or (2) insert an element, using the same assumptions as above, is

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Putting this into tabular form, we get the following:

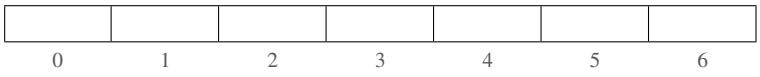
α	Average Probes Needed for Successful Search	Average Probes Needed for Unsuccessful Search
0.1	1.1	1.1
0.2	1.1	1.3
0.3	1.2	1.5
0.4	1.3	1.9
0.5	1.5	2.5
0.6	1.8	3.6
0.7	2.2	6.1
0.8	3.0	13.0
0.9	5.5	50.5

Notice that the number of probes we need worsens as the table gets more and more full. In general, if we are implementing a hash table with linear probing, we do not want the table to be more than half full.

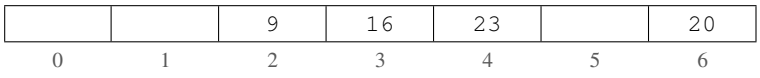
The size of the hash table plays a greater importance if quadratic probing is used instead of linear probing. This is because of a mathematical property related to *quadratic residues*, or the set of possible remainders of square numbers. If we are given the equation

$$(c + j^2) \bmod M$$

for any constant c, collision number j, and table size M, *we can only produce a limited number of remainders upon division.* Consider the hash function $h(k) = k$ for a hash table of size 7, where quadratic probing is used. We want to insert the keys 9, 16, 20, 23, and 30.



After inserting 9, 16, 20, and 23 using quadratic probing, we would get the following:



Now, let's try inserting 30. Upon first glance, we would initially expect 30 to fall into index 0, 1, or 5. However, 30 hashes to index 2, which is already occupied. We would then check index $(2 + 1^1) \bmod 7 = 3$, which is also full. Then, we would check index $(2 + 2^2) \bmod 7 = 6$, which is also full. In fact, if we keep on going:

- $(2 + 3^2) \bmod 7 = 4$ (full)
- $(2 + 4^2) \bmod 7 = 4$ (full)
- $(2 + 5^2) \bmod 7 = 6$ (full)
- $(2 + 6^2) \bmod 7 = 3$ (full)
- $(2 + 7^2) \bmod 7 = 2$ (full)
- $(2 + 8^2) \bmod 7 = 3$ (full)
- …

We have encountered a key that causes infinite collisions! This is because a modulus of 7 only supports four distinct quadratic residual values, so there is no way to get a new unoccupied index for the fifth key if we keep our table size at 7. In fact, for any table size M, approximately only half of the integers between 1 and M are valid quadratic residual values mod M. Thus, if quadratic probing is used, it can be dangerous to have a table that is over half full, since you may end up with a key that cannot be inserted into the table at all.

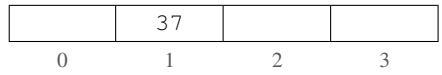
As the load factor increases, the performance of open addressing techniques degrades quickly. This is one of the big disadvantages of using an open addressing technique instead of separate chaining to resolve collisions. For separate chaining, the search time increases gradually if you increase the number of keys in the table without adjusting the table size. If you double the number of keys using separate chaining, for example, you would expect the average list length at each index to double, assuming that keys are evenly distributed. However, for open addressing, the search time increases dramatically as the table fills (to a point where no more keys can be inserted when $\alpha = 1$). This is because each cell of the table can only hold one element, and it becomes harder to find an open position as the table fills up.

※ 17.4.2 *Dynamic Hashing*

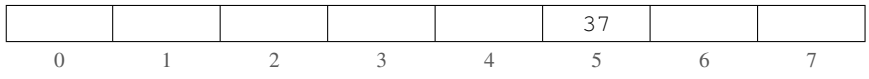
Nonetheless, a hash table's performance degrades if we increase the number of keys without adjusting the size of the table, regardless of whether we use separate chaining or open addressing. Thus, to maintain a hash table's efficiency, we will need to dynamically resize the hash table based on the number of keys in the table. This process is known as **dynamic hashing**. A common dynamic hashing procedure is to

- Double the size of the hash table if the load factor ever exceeds 0.5.
- Rehash every element in the old table to the new table. (Note that "deleted" positions don't store valid elements, so they aren't rehashed.)

The rehashing step is important because elements may not have the same position when they are moved to a new table. For example, if we had a hash table of size 4 that compresses keys using $c(k) = k \bmod M$, the key 37 would go into index 37 mod 4 = 1.



However, if we double the hash table to size 8, the key 37 would instead fall into index 37 mod 8 = 5 instead.



As a result, we cannot directly copy 37 from index 1 of the old table to index 1 of the new table, since 37 would no longer hash to that position. Instead, we need to rehash 37 to obtain its position in the new table before adding it in.

This process is expensive, but it is also infrequent. Using amortized analysis, we can actually prove that insertion into a hash table takes amortized constant time, even if a single insertion may require elements to be rehashed into a larger table. Suppose we have a hash table that doubles its size if $\alpha > 0.5$ and rehashes all existing elements over. Assuming the cost of inserting an element is 1, the following table shows the work associated with each insertion, starting from an empty table:

Item Number	1	**2**	**3**	4	**5**	6	7	8	**9**	10	11	12	13	14	15	16
Table Size	2	**4**	**8**	8	**16**	16	16	16	**32**	32	32	32	32	32	32	32
Cost	1	**1+1**	**1+2**	1	**1+4**	1	1	1	**1+8**	1	1	1	1	1	1	1

The load factor exceeds 0.5 on insertions 2, 3, 5, and 9, which require us to copy the existing elements over to a new, larger table. This is why the shaded insertions require a cost larger than 1 (e.g., on insertion 9, we have to rehash the original 8 elements before adding the 9$^{\text{th}}$ element, for a total cost of 9). The total work required to insert n elements can thus be expressed as

$$T(n) = \underbrace{(1+1+1+1+\ldots)}_{\text{cost of inserting new elements}} + \underbrace{(1+2+4+8+\ldots+n')}_{\text{cost of reallocation to new table}}$$

where $1+1+\ldots$ is done n times, and n' represents the largest power of two that is smaller than n. Since 1 is added n times, the total cost of appending new elements is n. To determine the cost of copying during reallocation, we can use the following identity

$$\sum_{k=0}^{n-1}(ar^k) = a\left(\frac{1-r^n}{1-r}\right)$$

where a is the first term, r is the common ratio, and n is the number of terms. We can express $1+2+4+8+\ldots+n'$ as

$$\sum_{k=0}^{(\log_2 n+1)-1} 2^k = \frac{1-2^{\log_2 n+1}}{1-2} = \frac{1-2^{\log_2 n}\times 2}{1-2} = \frac{1-2n}{1-2} = \frac{2n-1}{2-1} = 2n-1$$

Putting it all together, the total work required to push n elements in the worst case is bounded by

$$T(n) = n + 2n - 1 = \Theta(n)$$

The amortized cost of inserting into this hash table is therefore

$$\text{Amortized Complexity} = \frac{T(n)}{n} = \frac{\Theta(n)}{n} = \Theta(1)$$

※ 17.4.3 *Summary of Collision Resolution Techniques, Load Factor, and Dynamic Hashing*

In summary, there are four different ways to resolve a collision if two keys end up hashing to the same position. One method, separate chaining, creates a linked list for each index of the table. The other three methods — linear probing, quadratic probing, and double hashing — utilize a process known as open addressing, which looks for another empty index in the table to insert the new value. Assuming the collision occurred at index i, linear probing probes for open positions sequentially starting from the collision index (i.e., $i+1$, $i+2$, …), quadratic probing probes for open positions at square distances from the collision index (i.e., $i+1^2$, $i+2^2$, …), and double hashing applies an additional hash function t' to determine how far from the original hash value to probe (i.e., $i+1\times t'(key)$, $i+2\times t'(key)$, …). In addition, the load factor of a hash table, denoted by $\alpha = N/M$, is a value that can be used to reflect a table's efficiency. The larger the load factor, the less efficient a hash table is. Thus, to ensure that the performance of a hash table remains optimal at all times, we utilize a process known as dynamic hashing, which increases the table size whenever the load factor exceeds a certain threshold.

17.5 The STL Unordered Map Container

The STL implements two important containers using hashing: the `std::unordered_map<>` and the `std::unordered_set<>`. The `std::unordered_map<>`, in the `<unordered_map>` library, is an associative container that stores key-value pairs. The key is used to *uniquely* identify an element, and the value is used to store information associated with each key. Unordered maps are useful for problems that require fast lookup of data given a key (e.g., looking up a student's records given their ID). Consider the food prices that we introduced earlier:

Food Item	Price
Apple	$3.99
Avocado	$4.99
Banana	$2.49
Flour	$2.29
Ginger	$2.69
Milk	$2.09
Tofu	$3.79

If we wanted to store this data in a way that allows us to efficiently retrieve the price of any food item given its name, we could put this in an `std::unordered_map<>`. To create one, we must pass in the types of the key and mapped value, as follows:

```
std::unordered_map<KEY_TYPE, VALUE_TYPE> map_name;
```

In this problem, we are given the name of a food as a string, and we want to use this name to look up its price. Thus, the key has a type of `std::string`, and the mapped value has a type of `double`.

```
std::unordered_map<std::string, double> food_prices;
```

We can then go through and insert our data into the `std::unordered_map<>`:

```
food_prices.insert({"Apple", 3.99});
food_prices.insert({"Avocado", 4.99});
food_prices.insert({"Banana", 2.49});
food_prices.insert({"Flour", 2.29});
food_prices.insert({"Ginger", 2.69});
food_prices.insert({"Milk", 2.09});
food_prices.insert({"Tofu", 3.79});
```

After inserting the key-value pairs, `operator[]` can be used to retrieve the value associated with each key. For instance, if you wanted to get the price of "Tofu", you can just call `food_prices["Tofu"]` (you can think of this as "indexing" into the unordered map using the key).

```
double price_tofu = food_prices["Tofu"];
std::cout << price_tofu << '\n';  // prints 3.79
```

The following lists some common member functions that can be used with an `std::unordered_map<>`:

```
template <typename K, typename V>
std::pair<iterator, bool> std::unordered_map<K, V>::insert(const std::pair<K, V>& p);
```

Attempts to insert a key-value pair into the container. Since keys in an `std::unordered_map<>` must be unique, an insertion is done only if the key does not already exist in the container. Returns a pair consisting of an iterator to the inserted element (or the element that prevented insertion) and a `bool` for whether the insertion took place.

```
template <typename K, typename V, typename... Args>
std::pair<iterator, bool> std::unordered_map<K, V>::emplace(Args&&... args);
```

Attempts to insert a key-value pair into the container, constructed in-place using `args`. Since keys in an `std::unordered_map<>` must be unique, an insertion is done only if the key does not already exist in the container. Returns a pair consisting of an iterator to the inserted element (or the element that prevented insertion) and a `bool` for whether the insertion took place.

```
1  std::unordered_map<std::string, double> food_prices;
2  auto item1 = food_prices.insert({"Apple", 3.99});
3  std::cout << item1.second << '\n';  // prints 1 for true
4  auto item2 = food_prices.insert({"Apple", 3.89});
5  std::cout << item2.second << '\n';   // prints 0 for false (key "Apple" exists)
6  std::cout << food_prices["Apple"] << '\n';  // prints 3.99 (first value inserted)
7
8  // emplace does not require you to pass in a pair, you can pass in constructor args instead
9  auto item3 = food_prices.emplace("Banana", 2.49);
10 std::cout << item3.second << '\n';  // prints 1 for true
```

C++17 introduced the `try_emplace()` member method, which can be used to perform insertions more efficiently for certain key-value pairs. As an example, the `insert()` and `emplace()` methods covered above may involve the construction of a key-value pair when invoked, even if doing so is not necessary. In contrast, the `try_emplace()` method does *not* touch its provided arguments if the key already exists in the table. This optimization is particularly useful if the value object is expensive to construct.

```
template <typename K, typename V, typename... Args>
std::pair<iterator, bool> std::unordered_map<K, V>::try_emplace(K&& k, Args&&... args);
std::pair<iterator, bool> std::unordered_map<K, V>::try_emplace(const K& k, Args&&... args);
```

Inserts a new element into the container with key `k` and value constructed using `args`, but only if `k` does not exist as a key in the table already. Unlike `insert()` and `emplace()`, `try_emplace()` does not move from rvalue arguments if an insertion does not happen, which makes them useful when working with maps whose values are move-only types such as `std::unique_ptr<>` (covered in chapter 27). Also, unlike `emplace()`, `try_emplace()` treats the arguments of the key and mapped value separately, so you do not need the constructor arguments to directly construct a `std::pair<>` (i.e., `try_emplace(key, value_arg1, value_arg2, ...)` works instead of `emplace(key, value)` which passes in an instance of the value directly, while `value_arg...` can be used to construct `value` in-place). The return value is the same as that of `emplace()`.

```
std::unordered_map<std::string, BigObject> big_objects;
std::string k = "key";

// insert key k, so k is now in the hash table with a BigObject value
// all subsequent insertions with key k should fail to insert a new key-value pair
big_objects[k] = BigObject(arg1, arg2, arg3, ...);

// constructs a pair with BigObject that ends up being immediately destructed since k in table
big_objects.insert({k, BigObject(arg1, arg2, arg3, ...)});

// constructs a pair with BigObject that ends up being immediately destructed since k in table
big_objects.emplace(k, BigObject(arg1, arg2, arg3, ...));

// does not construct a BigObject at all since k is already in the table (optimal)
big_objects.try_emplace(k, arg1, arg2, arg3, ...);
```

Remark: Because the insert operations for an `std::unordered_map<>` return a pair containing an iterator and a Boolean indicating if the insertion was successful, C++17's structured bindings can be useful in extracting this return value in a clean manner (if you want a refresher on structured bindings, see section 11.13.1). An example is shown below:

```
std::unordered_map<std::string, double> food_prices;
auto [it, success] = food_prices.try_emplace("Apple", 3.99);
std::cout << it->first << '\n';    // prints "Apple"
std::cout << it->second << '\n';   // prints "3.99"
it->second = 4.99;                 // change price of "Apple" to 4.99
std::cout << success << '\n';      // prints "1" for true

// this insertion does not do anything since "Apple" is already in table
auto [it2, success2] = food_prices.try_emplace("Apple", 3.99);
std::cout << it2->first << '\n';   // prints "Apple"
std::cout << it2->second << '\n';  // prints "4.99" (from previous update)
std::cout << success2 << '\n';     // prints "0" for false
```

```
template <typename K, typename V>
bool std::unordered_map<K, V>::empty();
```

Returns a `bool` indicating whether the unordered map is empty.

```
std::unordered_map<std::string, double> food_prices;
std::cout << food_prices.empty() << '\n';  // prints true
```

```
template <typename K, typename V>
size_t std::unordered_map<K, V>::size();
```

Returns the number of elements in the container.

```
std::unordered_map<std::string, double> food_prices;
food_prices.insert({"Apple", 3.99});
food_prices.insert({"Avocado", 4.99});
std::cout << food_prices.size() << '\n';  // prints 2
```

`template <typename K, typename V>` `iterator std::unordered_map<K, V>::begin();`
Returns an iterator pointing to the first element in the `std::unordered_map<>` container (`.cbegin()` returns a `const` version of this iterator).
`template <typename K, typename V>` `iterator std::unordered_map<K, V>::end();`
Returns an iterator pointing to one past the last element in the `std::unordered_map<>` container (`.cend()` returns a `const` version of this iterator).

The iterators returned by `begin()` and `end()` point to a key-value pair. Note that an `std::unordered_map<>` is unordered, and thus *does not provide a guarantee on which element is considered first*. However, if you iterate from the `begin()` iterator to the `end()` iterator, you are guaranteed to visit every element in the `std::unordered_map<>` (however, the order in which elements are visited is non-deterministic).

```
1   std::unordered_map<std::string, double> food_prices;
2   food_prices.insert({"Apple", 3.99});
3   auto it = food_prices.begin();
4   std::cout << it->first << '\n';   // prints Apple
5   std::cout << it->second << '\n';  // prints 3.99
```

`template <typename K, typename V>` `void std::unordered_map<K, V>::clear();`
Erases all elements in the container, dropping size to 0.
`template <typename K, typename V>` `iterator std::unordered_map<K, V>::erase(const iterator pos);`
Erases the element pointed to by `pos` and returns an iterator to the element following the one that was removed.
`template <typename K, typename V>` `iterator std::unordered_map<K, V>::erase(const iterator first, const iterator last);`
Erases all elements in the iterator range `[first, last)` and returns an iterator to the element following the last element removed.
`template <typename K, typename V>` `size_t std::unordered_map<K, V>::erase(const K& key);`
Erases the element with the key equivalent to `key` and returns the number of elements removed (which should always be 1 for an `std::unordered_map<>` since keys are unique).

```
1   std::unordered_map<std::string, double> food_prices;
2   food_prices.insert({"Apple", 3.99});
3   food_prices.insert({"Avocado", 4.99});
4   food_prices.erase("Apple");
5   std::cout << food_prices.size() << '\n';  // prints 1
```

`template <typename K, typename V>` `V& std::unordered_map<K, V>::operator[](const K& key);`
Returns a reference to the value associated with the given key. **If the key does not already exist, it will be inserted into the container. When this happens, the value associated with this key is value-initialized.**

It is important to remember that `operator[key]` creates the key if it does not already exist! As a result, `operator[KEY] = VALUE` can be used as an alternative to `.insert()` to add values into an `std::unordered_map<>`:

```
1   std::unordered_map<std::string, double> food_prices;
2   food_prices["Apple"] = 3.99;
3   food_prices["Avocado"] = 4.99;
4   food_prices["Banana"] = 2.49;
5   food_prices["Flour"] = 2.29;
6   food_prices["Ginger"] = 2.69;
7   food_prices["Milk"] = 2.09;
8   food_prices["Tofu"] = 3.79;
```

However, this behavior can also lead to some unexpected consequences. Consider the following `std::unordered_map<>` that stores the number of days in a month.

```
1   std::unordered_map<std::string, int32_t> days_in_month;
2   days_in_month["January"] = 31;
3   days_in_month["February"] = 28;
4   days_in_month["March"] = 31;
5   days_in_month["April"] = 30;
6   days_in_month["May"] = 31;
7   days_in_month["June"] = 30;
8   days_in_month["July"] = 31;
9   days_in_month["August"] = 31;
10  days_in_month["September"] = 30;
11  days_in_month["October"] = 31;
12  days_in_month["November"] = 30;
13  days_in_month["December"] = 31;
```

Suppose this `std::unordered_map<>` is used in a program that takes in a month from the user and prints out the number of days in that month. The following implementation would *not* be fully correct:

```
std::string input_month;
std::cout << "Enter a month name: ";
std::cin >> input_month;

// print out the number of days
std::cout << input_month << " has " << days_in_month[input_month] << " days\n";
```

If the user correctly inputs a month, this would run with no errors. However, what if the user entered a bad month name, like `"Feb"` or `"Banana"`? `operator[]` would attempt to look up those keys, but they do not exist in the table. As a result, the operator would create an element in the table with the key `"Feb"` or `"Banana"` with a value that is value-initialized to zero — this is not something you want! Instead, if you want to retrieve the value of a key that you are not sure actually exists, you will need to use the `.find()` member function first.

```
template <typename K, typename V>
iterator std::unordered_map<K, V>::find(const K& key);
```
Checks if `key` exists in the container. If it exists, the function returns an iterator to the element with that key. If it does not exist, the function returns an iterator that points one past the end (i.e., the `end` iterator).

The correct way to look up an item is to first check if it exists by making sure that `.find()` does not return the `end` iterator. You should only use `operator[]` if you are certain that a key exists in the table. This prevents non-existent keys from being added to the `std::unordered_map<>` without your awareness.

```
auto it = months.find(input_month);
if (it == months.end()) {
  std::cout << input_month << " not found\n";
} // if
else {
  std::cout << it->first << " has " << it->second << " days\n";
} // else
```

Since iterators to an element in the `std::unordered_map<>` point to a key-value pair, `it->first` represents the key and `it->second` represents the value stored at that key. Furthermore, because `.find()` returns an iterator to a key if it exists, you do *not* need to make another lookup to obtain the value of that key!

An alternative to `.find()` is `.count()`, which returns the number of elements with a given key. Since an `std::unordered_map<>` does not allow duplicate keys, this function will always return either 0 or 1 (0 if the key does not exist, 1 if it does).

```
template <typename K, typename V>
size_t std::unordered_map<K, V>::count(const K& key);
```
Returns the number of elements whose key compares equal to `key`. For an `std::unordered_map<>`, this function returns 0 if the key does not exist, and 1 if it does.

```
if (months.count(input_month)) {
  std::cout << input_month << " has " << months[input_month] << " days\n";
} // if
else {
  std::cout << input_month << " not found\n";
} // else
```

However, `.find()` is preferred since it returns an iterator to the actual element that matches the given key. As a result, you can just use the iterator to access the value without having to make a second lookup. With `.count()`, you may need to make two lookups if the key exists: one during the `.count()` call to determine if the key exists, and one using `operator[]` to actually retrieve the value associated with that key.

Behind the scenes, the STL `std::unordered_map<>` is implemented as a hash table that uses separate chaining (with linked lists) to resolve collisions. This is due to the requirements of the C++ standard, which make it difficult to implement collision resolution in any other way. However, you may encounter different implementations of hash tables that use other collision resolution methods if you ever work for a company that has their own custom implementation.

Since an `std::unordered_map<>` relies on hashing for performance, the key of a hash table must be hashable (i.e., there must exist a hash function that can convert an object of type `key_type` into an index). For many pre-defined types like `int`, `double`, and `std::string`, a hash function is provided for you. However, if you want to create an `std::unordered_map<>` where your key has a custom type, you will need to define a custom hash function for that type (which will be discussed in more detail in section 17.8).

Complexity of `std::unordered_map<>` Operations

Insert Element	Access Element	Erase Element	Find Element
average-case $\Theta(1)$	average-case $\Theta(1)$	average-case $\Theta(1)$	average-case $\Theta(1)$
worst-case $\Theta(n)$	worst-case $\Theta(n)$	worst-case $\Theta(n)$	worst-case $\Theta(n)$

Example 17.5 Write a function that takes in a text file in the form of a stream and prints out the word count for each word in the stream. For example, given a stream with the contents:

```
eecs is fun is it
```

You would print out the following:

```
eecs 1
is 2
fun 1
it 1
```

The words may be printed out in any order. You may assume that all the words in the stream are in lower case, and there is no punctuation present. The function header is shown below:

```
void print_word_count(std::istream& words);
```

For this problem, we are given a collection of words, and we want to identify the word count for each word. This fits perfectly with the key-value structure that is ideal for a `std::unordered_map<>`, since we are given a key (a word) and an associated value that we need to lookup for that key (the word count). To solve this problem, we will create a `std::unordered_map<>` that maps each word to its word count. Then, we will read in each of the words and increment its count in the hash table. The code is shown below:

```
1  void print_word_count(std::istream& words) {
2    std::unordered_map<std::string, int32_t> word_count_map;
3    std::string current_word;
4    while (words >> current_word) {
5      ++word_count_map[current_word];
6    } // while
7    for (auto& word_pair : word_count_map) {
8      std::cout << word_pair.first << " " << word_pair.second << '\n';
9    } // for word_pair
10 } // print_word_count()
```

On line 5, we do not check if the word already exists in the `std::unordered_map<>` before we use `operator[]`; as a result, if `current_word` does not already exist in the hash table, it gets added automatically (with a word count that is value-initialized to 0). For this specific problem, this is intended behavior. However, this is not always the case, and using `operator[]` on non-existent keys is dangerous if you do not want to automatically add these keys to the table.

If you iterate through an `std::unordered_map<>`, each element that you visit is actually a key-value pair , so you will have to call `.first` to obtain the key, and `.second` to obtain the value (or use a structured binding). This is shown on line 7 (although you can just use `auto` if you do not want to explicitly write out the type). In addition, because keys in an `std::unordered_map<>` are unordered, you cannot guarantee the order in which words are printed out. If you want to print out the words in a predetermined order, you will have to rely on a different data structure that supports this functionality (one possible alternative is a `std::map<>`, which will be covered in the next chapter).

Because `operator[]` returns a reference to the value associated with a given key, you are allowed to do things like this:

```
1  struct Student {
2    std::string uniqname;
3    std::string full_name;
4    std::vector<double> grades;
5  };
6
7  std::unordered_map<std::string, Student> all_students;
8
9  void add_grade(const std::string& uniqname, double grade) {
10   all_students[uniqname].grades.push_back(grade);
11 } // add_grade()
```

On line 10, `all_students[uniqname]` is a reference to the `Student` object associated with the key `uniqname`, which is why we were able to directly access `all_students[uniqname].grades`, which is a vector. This makes it easy to retrieve and work with the value of a key without having to explicitly store it somewhere else. However, you have to be careful not to call `operator[]` on the same key over and over again if you do not need to, since every call to `operator[]` requires a lookup (which hashes the key). Consider the following code:

```
1  struct Employee {
2    std::string title;
3    double salary;
4    int32_t years_working;
5  };
6
7  std::unordered_map<std::string, Employee> employee_map;
8
9  void update_employee_info(const std::string& name, const std::string& title,
10                            double salary, int32_t years_working) {
11   employee_map[name].title = title;
12   employee_map[name].salary = salary;
13   employee_map[name].years_working = years_working;
14 } // update_employee_info()
```

This code ends up making three lookups (on lines 11, 12, and 13), when only one lookup is necessary. Although each lookup takes constant time, the work required to hash and look up each key is non-negligible and can quickly accumulate. The alternative is to only make one lookup and store an iterator or reference to the value you want to use (the following code assumes that `name` is guaranteed to be a key in the table).

```
1  // Method 1: store an iterator (only one lookup needed)
2  auto it = employee_map.find(name);
3  it->second.title = title;
4  it->second.salary = salary;
5  it->second.years_working = years_working;
6
7  // Method 2: store a reference (only one lookup needed)
8  auto& employee = employee_map[name]; // make sure to include the ampersand
9  employee.title = title;
10 employee.salary = salary;
11 employee.years_working = years_working;
```

This is particularly true if you are trying to check the existence of a key before you use it. From a performance standpoint, you should avoid doing something like this:

```
1  if (my_map.find(key) != my_map.end()) {
2    std::cout << "Found: " << my_map[key] << '\n';
3  } // if
```

Notice that the code above makes two lookups instead of one: once with the `.find()` call on line 1 and once with the `operator[]` call on line 2. To fix this, use the iterator returned by `.find()` instead of calling `operator[]` again:

```
1  auto it = my_map.find(key);
2  if (it != my_map.end()) {
3    std::cout << "Found: " << iter->second << '\n';
4  } // if
```

With this change, only a single lookup is made.

Remark: The `std::unordered_map<>` is a useful data structure that can be used to solve many different types of problems. However, a hash table may not be the best container to use depending on the problem at hand. If your keys are small integers, it may be better to use an vector and use direct addressing to look up values associated with each key. In general, if a problem can be efficiently solved without a hash table, then it may make sense not to use one. Hash tables are extremely versatile, but they also require significantly higher memory overhead and sophisticated operations to support their functionality (e.g., computing a hash for every key). Depending on the problem you are trying to solve, an overreliance on hash tables may end up making your time and memory performance worse!

17.6 The STL Unordered Set Container

The STL also provides the `std::unordered_set<>` container, which can be found in the `<unordered_set>` library. Like an `std::unordered_map<>`, an `std::unordered_set<>` stores *unique* elements in no particular order. However, one major difference with a `std::unordered_set<>` is that the value of an element is at the same time treated as its key. Much like its name implies, an `std::unordered_set<>` is a good container to use if you want a way to efficiently keep track of whether an item exists in a set.

Similar to the STL's `std::unordered_map<>`, the `std::unordered_set<>` is implemented using a hash table as its underlying structure, with separate chaining as the collision resolution technique. An `std::unordered_set<>` also supports many of the same operations as an `std::unordered_map<>`, with the exception of `operator[]` (as there is no mapping from a key to a separate value). The complexities of `std::unordered_set<>` operations are summarized below:

Insert Element	Access Element	Erase Element	Find Element
average-case $\Theta(1)$	average-case $\Theta(1)$	average-case $\Theta(1)$	average-case $\Theta(1)$
worst-case $\Theta(n)$	worst-case $\Theta(n)$	worst-case $\Theta(n)$	worst-case $\Theta(n)$

A few common `std::unordered_set<>` operations are summarized below.

```
template <typename K>
std::pair<iterator, bool> std::unordered_set<K>::insert(const K& key);
```
Attempts to insert a key into the container. Since keys in an `std::unordered_set<>` must be unique, an insertion is done only if the key does not already exist in the container. Returns a pair consisting of an iterator to the inserted element (or the element that prevented insertion) and a `bool` for whether the insertion took place.

```
template <typename K, typename InputIterator>
void std::unordered_set<K>::insert(InputIterator first, InputIterator last);
```
Attempts to insert all elements from the range `[first, last)` into the `std::unordered_set<>`.

```
template <typename K>
void std::unordered_set<K>::insert(std::initializer_list<K> ilist);
```
Attempts to insert all the elements from the initializer list `ilist` into the `std::unordered_set<>` (for example, the line `my_set.insert({1, 3, 5, 7})` inserts the elements 1, 3, 5, and 7).

```
std::unordered_set<int32_t> my_set;
my_set.insert(280);
my_set.insert(281);
my_set.insert(370);
my_set.insert(376);

std::vector<int32_t> vec = {481, 482, 483, 484, 485, 486};
my_set.insert(vec.begin(), vec.end());
// my_set now stores {280, 281, 370, 376, 481, 482, 483, 484, 485, 486} (may NOT be in this order)
```

`template <typename K>` `bool std::unordered_set<K>::empty();`
Returns a `bool` indicating whether the unordered set is empty.
`template <typename K>` `size_t std::unordered_set<K>::size();`
Returns the number of elements in the container.
`template <typename K>` `iterator std::unordered_set<K>::begin();`
Returns an iterator pointing to the first element in the `std::unordered_set<>` container (`.cbegin()` returns a `const` version of this iterator).
`template <typename K>` `iterator std::unordered_set<K>::end();`
Returns an iterator pointing to one past the last element in the `std::unordered_set<>` container (`.cend()` returns a `const` version of this iterator).
`template <typename K>` `void std::unordered_set<K>::clear();`
Erases all elements in the container, dropping size to 0.
`template <typename K>` `iterator std::unordered_set<K>::erase(const iterator pos);`
Erases the element pointed to by `pos` and returns an iterator to the element following the one that was removed.
`template <typename K>` `iterator std::unordered_set<K>::erase(const iterator first, const iterator last);`
Erases all elements in the iterator range `[first, last)` and returns an iterator to the element following the last element removed.
`template <typename K>` `size_t std::unordered_set<K>::erase(const K& key);`
Erases the element with the key equivalent to `key` and returns the number of elements removed (which is always 1 since keys are unique).
`template <typename K>` `iterator std::unordered_set<K>::find(const K& key);`
Checks if `key` exists in the container. If it exists, the function returns an iterator to the element with that key. If it does not exist, the function returns an iterator that points one past the end (i.e., the `end` iterator).
`template <typename K>` `size_t std::unordered_set<K>::count(const K& key);`
Returns the number of elements whose key compares equal to `key`. For an `std::unordered_set<>`, this function returns 0 if the key does not exist, and 1 if it does.

Example 17.6 Given a vector of integers, write a program that finds the first repeated integer in the container. If there are no repeated integers in the vector, return `-1`.

Since this problem is asking us to determine if we have seen an element before, an `std::unordered_set<>` would be an appropriate container to use (for this example, there is no need to map a key to a value). To solve this problem, we would initialize an `std::unordered_set<>` and iterate through the vector. For each element, we would first check if it already exists in our set. If it exists, we have found the first repeated integer and would return that value. Otherwise, we would add the value into the set and continue looking. The solution code is shown below:

```
int32_t first_repeated_element(const std::vector<int32_t>& vec) {
  std::unordered_set<int32_t> seen;
  for (int32_t val : vec) {
    auto it = seen.find(val);
    if (it == seen.end()) {    // element not found
      seen.insert(val);
    } // if
    else {                     // element seen before
      return *it;
    } // else
  } // for val
  return -1;
} // first_repeated_element()
```

The worst-case time complexity of this solution is $\Theta(n)$, where n is the size of the vector of integers.

17.7 The STL Unordered Multimap and Unordered Multiset Containers (*)

The STL also provides an `std::unordered_multimap<>` container, which is an unordered map that supports duplicate keys. Since the same key can exist more than once in these containers, a single key may be mapped to multiple values. Thus, to retrieve the value of a key in an `std::unordered_multimap<>`, you have to iterate through all elements in the multimap that share that key. To retrieve all the elements that share a given key, you can use the `.equal_range()` member function:

```
template <typename K, typename V>
std::pair<iterator, iterator> std::unordered_multimap<K, V>::equal_range(const K& key);
```

Returns an iterator range containing all elements in the multimap with the given key. Like with other STL algorithms, the first iterator returned is inclusive, and the second iterator returned is exclusive.

For example, the following multimap maps department names to the classes provided under that department.

```
1  std::unordered_multimap<std::string, int32_t> classes;
2  classes.insert({"EECS", 280});
3  classes.insert({"MATH", 217});
4  classes.insert({"EECS", 183});
5  classes.insert({"ENGR", 101});
6  classes.insert({"EECS", 281});
```

If you want to retrieve all values in the multimap that have the key `"EECS"`, you can call `.equal_range("EECS")` to get an iterator range that includes all elements in the multimap with that key. You would then have to iterate through the range and visit each value individually.

```
7  auto iter_range = classes.equal_range("EECS");
8  for (auto it = iter_range.first; it != iter_range.second; ++it) {
9    std::cout << it->first << " " << it->second << '\n';
10 } // for it
```

Since the `std::unordered_multimap<>` is unordered, you cannot guarantee the order of elements in the iterator range. The output of the above code could be the following (these three lines need not be printed in this order):

```
EECS 281
EECS 183
EECS 280
```

However, for this course, there will never be a need to use this container. If you want a key to map to multiple values, you can just use a standard `std::unordered_map<>` and map each key to a container of values. This gives you greater freedom and flexibility when working with the values associated with each key and removes the risk of non-deterministic behavior (like the scenario above). For example, the following code does the same thing as a the code above, but without the need for a multimap:

```
1  // map each string to a vector of ints
2  std::unordered_map<std::string, std::vector<int32_t>> classes;
3  classes["EECS"].push_back(280);
4  classes["MATH"].push_back(217);
5  classes["EECS"].push_back(183);
6  classes["ENGR"].push_back(101);
7  classes["EECS"].push_back(281);
8  // print out all values associated with "EECS"
9  auto it = classes.find("EECS");
10 if (it != classes.end()) {
11   for (int32_t class_num : it->second) {
12     std::cout << it->first << " " << class_num << '\n';
13   } // for class_num
14 } // if
```

The `std::unordered_multiset<>` is another container that is provided by the STL. As its name implies, this container behaves like an `std::unordered_set<>` but allows for duplicate keys. Much like before, the `std::unordered_multiset<>` relies on the `.equal_range()` member function to return an iterator range to all instances of a key that exists within the multiset.

Remark: If you are curious about the STL's `std::unordered_multimap<>` and `std::unordered_multiset<>` containers, you can read their documentation. However, you will *not* be responsible for knowing how to use these containers for this class. This section simply provides a brief introduction to these containers to let you know they exist, but anything you can do with an unordered multimap or unordered multiset for this class can also be done with another container class.

17.8 STL Hashing and Composite Hash Functions

※ 17.8.1 *std::hash*

As mentioned earlier, hash functions for many pre-defined types, such as `int`, `double`, and `std::string`, are provided by the C++ standard library. The default hash function used by the standard library can be accessed using `std::hash<>`, a unary function object defined in the `<functional>` library. If you invoke the `std::hash<>` functor on an argument, you would get the hash value of that argument, using a default hash function provided by the standard library.

```
1  std::hash<std::string> hasher;           // hash function object on strings
2  std::cout << hasher("EECS281") << '\n';  // prints hash value of "EECS281"
```

The actual hash function itself is implementation-dependent. For instance, when the above hasher was run on the string `"EECS281"` in a CAEN environment, the hash value was 4147570959360960813. However, when the same code was run on Microsoft Visual Studio, the hash value was 3475765778. Nonetheless, these differences should not matter, since the hash function implemented by `std::hash<>` always hashes the same key to the same hash value regardless of what environment you are using (i.e., `"EECS281"` should always hash to 4147570959360960813 on CAEN using this hasher), and it also ensures that the probability of two different keys hashing to the same hash value is near zero.

※ 17.8.2 *Composite Hash Functions* (✻)

Hashing with pre-defined types is easy, since you can rely on hash functions that are provided by the standard library. However, what if you wanted to hash a custom object type? For instance, suppose you wanted to construct a hash table that maps coordinate points to location data, where the coordinates are defined as a custom object:

```
1  struct Coordinate {
2    int32_t x;
3    int32_t y;
4  };
5
6  std::unordered_map<Coordinate, std::string> location_map;
```

The above code would not compile, since `Coordinate` does not support a hash function, and thus cannot be the type of the key. If you wanted to use a `Coordinate` as the type of the key, you will have to define a custom hash function for the `Coordinate` object. One way to do so is to use a *composite hash function*, which combines the hash values of an object's components to calculate the hash value of the entire object itself. One example of a composite hash function is

$$H(\{x, y\}) = h(x) + p \times h(y)$$

where h is the hash function for each coordinate value, H is the composite hash function for the entire coordinate object, and p is an arbitrarily chosen integer. For instance, if $h(15) = 10$, $h(-27) = 20$, and $p = 5$, the hash value of the coordinate $(15, -27)$ would be $10 + 5 \times 20 = 110$.

However, a hash function that linearly combines the individual hash values of its components is not entirely ideal. This is because collisions using this hash function are easily predictable. If you want to write a hash function for a custom object, it is much better to combine the hash values of its components using the following equation instead of a linear combination.[2] Here, $h(k_i)$ is the hash value of component i of the custom object, and C is a constant (in the Boost library, C is the hexadecimal constant 0x9e3779b9, but you do not need to know why this number was chosen). *(You are not required to know the following material for this class!)*

$$\text{seed} \mathrel{\hat{}}= h(k_i) + C + (\text{seed} << 6) + (\text{seed} >> 2)$$

For example, if we started with an initial seed of 0, and $h(15) = 10$ and $h(-27) = 20$, the `Coordinate` object $(15, -27)$ would have a hash value of 175247765808:

1. Combine the hash value of k_1 with seed:

   0 ^= 10 + 0x9e3779b9 + (0 << 6) + (0 >> 2) = 2654435779

2. Combine the hash value of k_2 with seed:

   2654435779 ^= 20 + 0x9e3779b9 + (2654435779 << 6) + (2654435779 >> 2) = 175247765808

[2] This is the implementation of the hash combiner used by the Boost library, `boost::hash_combine`. However, Boost is prohibited in this class, so the examples on the following few pages implement this hash combiner as a separate function (instead of importing it from Boost).

Remark: What do the symbols ^=, << , and >> mean in the hash combiner equation? If you have never seen them before, don't worry! These are bit manipulation operators, which are covered in EECS 370 and not in this class. ^= is the *bitwise XOR assignment operator*, where `a ^= b` assigns `a` with the result of taking the XOR of `a` and `b`. To take the XOR (short for e**X**clusive **OR**) of two numbers, traverse the binary representation of the two numbers, compare the digits at each position, and return 1 if two bits are different and 0 if they are the same. For instance, the XOR of 281 (100011001 in binary) and 370 (101110010 in binary) is 107 (001101011 in binary):

$$\begin{array}{r} 100011001 \\ 101110010 \\ \hline 001101011 \end{array}$$

The << and >> might look like the insertion and extraction operators for streams, but in this case, they are *bit shifting operators*. If you apply a left shift (<<) on a number, you shift its underlying binary bit pattern to the left by one.

00000011	(3 in decimal)
00000110	(left shift, 6 in decimal)
00001100	(left shift, 12 in decimal)
00011000	(left shift, 24 in decimal)
00110000	(left shift, 48 in decimal)
01100000	(left shift, 96 in decimal)

Similarly, applying a right shift (>>) on a number shifts its underlying binary bit pattern to the right by one.

01011001	(89 in decimal)
00101100	(right shift, 44 in decimal)
00010110	(right shift, 22 in decimal)
00001011	(right shift, 11 in decimal)
00000101	(right shift, 5 in decimal)
00000010	(right shift, 2 in decimal)
00000001	(right shift, 1 in decimal)

In the previous formula, the term (seed << 6) is the value obtained after performing 6 left shifts on the value of seed, and the term (seed >> 2) is the value obtained after performing 2 right shifts on the value of seed.

Putting this all together, we can write the following code to generate the hash value of a `Coordinate` object:

```
1  void hash_combine(size_t& seed, const int32_t v) {
2    std::hash<int32_t> hasher;
3    seed ^= hasher(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
4  } // hash_combine()
5
6  size_t hash_coordinate(const Coordinate& coord) {
7    size_t hash_value = 0;
8    hash_combine(hash_value, coord.x);
9    hash_combine(hash_value, coord.y);
10   return hash_value;
11 } // hash_coordinate()
```

※ 17.8.3 *Hashing a Custom Type* (✻)

With this information, we can actually define `Coordinate` so that it can be accepted as the key-type of STL containers that utilize hashing, such as the `std::unordered_map<>` and `std::unordered_set<>`. To be able to use these containers with a custom key-type, we must define two things:

1. A **hash function** functor that overrides `operator()` and calculates the hash value of an object with the type of the key.
2. A **comparison function for equality** so that the hash table can compare instances of the custom-defined key in the case of a collision. This can be done by either (1) defining an equality predicate that overrides `operator()` or by (2) overloading `operator==` for the custom type (overloading `operator==` is typically the simpler method of the two).

Using the `Coordinate` object as an example, let's start by defining the comparison function for equality. We can do this by overloading `operator==` in the definition of the `Coordinate` type so that different `Coordinate` objects can be compared.

```
1  struct Coordinate {
2    int32_t x;
3    int32_t y;
4    bool operator==(const Coordinate& other) const {
5      return (x == other.x && y == other.y);
6    } // operator==()
7  };
```

Now, we will need to define a hash function for the `Coordinate` type. Using the composite hash function discussed earlier, we can write the following `CoordinateHasher` functor, which can be used to calculate the hash value of a `Coordinate` object by combining the hash values of its individual components.

```
9   struct CoordinateHasher {
10    void hash_combine(size_t& seed, const int32_t v) const {
11      std::hash<int32_t> hasher;
12      seed ^= hasher(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
13    } // hash_combine()
14
15    size_t operator()(const Coordinate& coord) const {
16      size_t hash_value = 0;
17      hash_combine(hash_value, coord.x);
18      hash_combine(hash_value, coord.y);
19      return hash_value;
20    } // operator()
21  };
```

After completing these steps, we can now use `Coordinate` as the key-type of an `std::unordered_map<>` or `std::unordered_set<>`. The following code will now compile (notice that we pass in `CoordinateHasher` as the third argument of the `std::unordered_map<>` template on line 24, which lets the map know to use `CoordinateHasher` to hash each coordinate):

```
23  int main() {
24    std::unordered_map<Coordinate, std::string, CoordinateHasher> location_map = {
25      {{-738, 277}, "Diag"},
26      {{-717, 291}, "Pierpont"},
27      {{-721, 294}, "Bursley"},
28      {{-716, 293}, "BBB"},
29      {{-738, 273}, "Ross"}
30    };
31
32    Coordinate c = {-717, 291};
33    auto it = location_map.find(c);
34    if (it != location_map.end()) {
35      std::cout << "The landmark at coordinates (" << c.x << ", " << c.y << ") is "
36                << it->second << '\n';
37    } // if
38  } // main()
```

The output of this code is:

```
The landmark at coordinates (-717, 291) is Pierpont
```

We can also apply a composite hash function in an `std::unordered_set<>`; this can be done by passing in the hash combiner functor as the second term of the container's initialization template. An example is shown below:

```
23  int main() {
24    std::unordered_set<Coordinate, CoordinateHasher> location_set;
25    Coordinate c = {-717, 291};
26    location_set.insert(c);
27    std::cout << location_set.count(c) << '\n';  // prints 1
28  } // main()
```

17.9 Solving Problems Using Hash Tables

Because they provide fast key-value lookup, hash tables are quite versatile and can be used to solve many different types of problems. In this section, we will go over a few problems that can be solved using a hash table container.

Example 17.7 Back in chapter 15, we introduced a problem where you are given an array of n positive integers that are sorted in ascending order, and you want to find a pair of two numbers in the array that sums to a given target number. We then introduced a $\Theta(n)$ time solution using the two pointer technique, where two indices are selectively incremented toward each other until the target sum is found.

Consider the same problem, but this time *the array of integers is no longer sorted in ascending order.* For example, given the array `[7, 10, 3, 9, 4]` and a target of `11`, you would return `[4, 7]`. Return `[-1, -1]` if there is no way to sum to the target number. Is it still possible to solve this problem in linear time, and what is the approach if it can?

Without the numbers being sorted, the two pointer technique can no longer be used, since it relies on knowledge on whether the pointers are moving toward a smaller or larger number. Because of this, we will need to find an alternative solution for this problem.

Notice that we want to find the solution to this problem in average-case linear time. This means that, if we iterate over the given array, the amount of work done at each element toward solving the problem must be constant on average. Therefore, if you want to store any information for each element, you would preferably want to store it in a constant-time lookup container like a hash table.

What information do we need to know at each element? Using our target value of 11, suppose we encountered 7 while iterating over our array. In this case, we would know that 7 is in the solution if and only if its complement of 11 - 7 = 4 is also in the array. However, we do not yet know if 4 is in the array or not, so we will need to store this information somewhere so that we can return the solution of `[4, 7]` if we ever encounter 4 in the array. This can be done using a hashing container that supports constant-time insertion and lookup, such as the `std::unordered_set<>`.

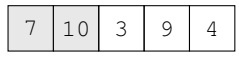
This forms the basis of our linear-time solution. We would iterate over the array one element at a time. At each element, we would push its complement (the value you get after subtracting the element from the target value) into an `std::unordered_set<>`. Then, if you ever encounter an element in the array that is in the `std::unordered_set<>`, you can simply return the corresponding pair. An illustration of this process is shown below using the example input array.

`std::unordered_set<>`
{}

The first element we consider in the array is 7. Since our target is 11, the complement of 7 is 11 - 7 = 4, so we insert 4 into our set.

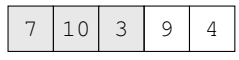
`std::unordered_set<>`
{4}

The next element in the array is 10. 10 is not in our set, so it is not part of a solution yet. We then insert 10's complement of 1 into the set.



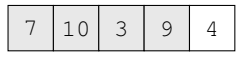
`std::unordered_set<>`
{4, 1}

The next element in the array is 3. 3 is not in our set, so it is not part of a solution yet. We then insert 3's complement of 8 into the set.



`std::unordered_set<>`
{4, 1, 8}

The next element in the array is 9. 9 is not in our set, so it is not part of a solution yet. We then insert 9's complement of 2 into the set.



`std::unordered_set<>`
{4, 1, 8, 2}

The next element in the array is 4. 4 is in our set, which means that we had encountered its complement in the array before. Thus, we know that 4 (along with its complement of 7) must be a valid pair that sums to our target, so we return `[4, 7]`.

`std::unordered_set<>`
{**4**, 1, 8, 2}

If we manage to iterate over the array without ever encountering a value that was already in the set, then there would be no solution.

Remark: If we had been asked to return the *indices* of the two values that sum to the target (instead of the values themselves), then we would need a way to keep track of the index of each value in the input array. This additional bit of information, however, does not alter our solution much; we can address this by using an `std::unordered_map<>` instead of an `std::unordered_set<>`, where each key is mapped to the index of its complement. Once we encounter a value in our map, we can also look up its complement's index in the map, thereby allowing us to easily return both indices in our solution. Our solution still runs in average-case linear time, as the `std::unordered_map<>` also supports average-case constant time insertion and lookup.[3]

An implementation of this solution is shown below:

```
1  std::pair<int32_t, int32_t> target_sum(const std::vector<int32_t>& vec, int32_t target) {
2    std::unordered_set<int32_t> complements;
3    for (int32_t val : vec) {
4      auto it = complements.find(val);
5      if (it == complements.end()) {
6        complements.insert(target - val);
7      } // if
8      else {
9        return {val, target - val};
10     } // else
11   } // for
12   return {-1, -1};  // no solution
13 } // target_sum()
```

The average-case time complexity of this solution is $\Theta(n)$, where n is the number of values in the input vector. This is because we are looping through the array once and then performing a lookup and/or an insertion on each item in the array (both of which take average-case constant time with a hash table container). The auxiliary space used by this solution is also $\Theta(n)$, since we are inserting complements into a separate container as we iterate over the array.

[3]This variation is actually a famous interview problem known as the *two-sum* problem, and you may see it mentioned as such from time to time.

Example 17.8 EECS 281 has had difficulty keeping up with the office hours demand this semester and has decided to limit the frequency that students can join the queue to *once every 60 minutes*. Unfortunately, we cannot code, so that is up to you! You are given a class object, *YamBot*, that handles all requests to join the queue. You must implement the class function `allow_to_join()` that takes in a timestamp (an integer in minutes) and a uniqname. If the student with that uniqname has joined the queue within the last 60 minutes, return `false`. Otherwise, return `true`. (*Note: This was a proposed written exam question for the Fall 2020 final exam.*)

Time Complexity: $\Theta(1)$ per call on average
Space Complexity: $\Theta(n)$ on average for making any number of calls with n different uniqnames

Example:

Uniqname	Timestamp	Explanation
joericha	0	Joe hasn't been on the queue yet, so return `true`
danlliu	35	Daniel hasn't been on the queue yet, so return `true`
joericha	40	Joe previously joined the queue at time 0, so return `false`
rushilk	50	Rushil hasn't been on the queue yet, so return `true`
joericha	70	Joe previously joined the queue at time 0, so his cooldown has finished, return `true`
danlliu	75	Daniel previously joined the queue at time 35, so return `false`
rushilk	110	Rushil previously joined the queue at time 50, so his cooldown has finished, return `true`

Starter Code:

```
class YamBot {
private:
  // TODO: Add any data structures here!
public:
  // Returns true if the student should be allowed to join the OH queue at
  // the given timestamp. Otherwise, the student will not be allowed to
  // join the queue, and you should return false. The timestamp is in minutes.
  bool allow_to_join(const std::string& uniqname, int32_t timestamp) {
    // TODO: Implement code here
  } // allow_to_join()
};
```

To successfully solve this problem, we would need to know the last timestamp at which each student joined the queue. Every time a student tries to join the queue, we would then compare the current timestamp with this most recent timestamp to determine if the student should be allowed into the queue again or not. This behavior fits well with the key-value lookup behavior supported by an `std::unordered_map<>`, especially since lookups can be done in average-case $\Theta(1)$ (which is required by the problem).

We can therefore implement this solution by using a `std::unordered_map<>` that maps each student's uniqname to the last time they joined the queue. Every time `allow_to_join()` is called on a student, we look up this student in the map to identify when they last joined the queue. There are three outcomes that may arise from this lookup:

- If the lookup fails to find the student's name, then the student never joined the queue before. We would therefore add their uniqname to the map with the current timestamp as the value and return `true`.
- If we find the student in the map, and their most recent timestamp was 60+ minutes ago, then their cooldown has finished and they can join the queue again. We would therefore update their most recent timestamp to the current time (since they joined the queue again) and return `true`.
- If we find the student in the map, and their most recent timestamp was less than 60 minutes ago, then they joined the queue too recently and should not be allowed to join again. Therefore, we return `false`.

An implementation of this solution is shown below:

```
class YamBot {
private:
  std::unordered_map<std::string, int32_t> oh_map;
public:
  bool allow_to_join(const std::string& uniqname, int32_t timestamp) {
    auto it = oh_map.find(uniqname);
    // first time joining the queue, add to map and return true
    if (it == oh_map.end()) {
      oh_map[uniqname] = timestamp;
      return true;
    } // if
    // cooldown finished, update timestamp and return true
    else if (timestamp - it->second >= 60) {
      it->second = timestamp;
      return true;
    } // else if
    // otherwise, joined less than 60 minutes ago, return false
    return false;
  } // allow_to_join()
};
```

Example 17.9 A *cache* is a temporary data storage location that holds frequently accessed data, allowing applications to speed up data retrieval. As an example, your browser may use a cache to store frequently accessed websites so that you do not have to refetch the contents of a website whenever you visit it; you can simply pull the data from a cache. This is beneficial because the contents of a cache are stored in memory that supports high-speed access (e.g., static random-access memory (SRAM), but this is an EECS 370 concept), thereby allowing you to access memory in a cache much faster than from anywhere else. Unfortunately, this type of memory is also expensive, so the capacity of the cache is bounded; you can only keep a limited amount of data in a cache at any point in time. Since the data that you want to work with may exceed the storage size of the cache, you will need a way to determine what data should be kept in the cache and what data should be removed if the cache capacity is reached. *(Note: None of the information is class material, just background for this problem.)*

One such method is to use a *LRU cache*, which uses a *least-recently used (LRU)* policy. In a LRU cache, the least recently used item is evicted from the cache first if the cache ever goes above its capacity. As an example, consider a cache with a capacity of 4 that stores the prices of items at a store (similar to the first example in this chapter). The following queries are made:

- Update price of apple to $3.99.
- Update price of banana to $2.49.
- Update price of ginger to $2.69.
- Update price of apple to $3.79.
- Update price of milk to $2.09.
- Update price of tofu to $3.79.
- Update price of ginger to $2.79.
- Update price of banana to $2.59.

We first update the price of apple. Since there is nothing in the cache yet, the key-value pair of `{"Apple", 3.99}` is added to the cache. Because it is the only value in the cache, it is also the least recently used value by default.

LRU Cache (Capacity 4)

Item	Price	LRU?
`"Apple"`	`3.99`	✓

Next, we update the price of banana. Since banana is not in the cache, we add it. The same applies for ginger, which is added next.

LRU Cache (Capacity 4)

Item	Price	LRU?
`"Apple"`	`3.99`	✓
`"Banana"`	`2.49`	
`"Ginger"`	`2.69`	

Next, we update the price of apple to $3.79. Notice that `"Apple"` is currently the least recently used element in the cache, but that would no longer be the case after the update. Instead, the least recently used element would become `"Banana"`.

LRU Cache (Capacity 4)

Item	Price	LRU?
`"Apple"`	`3.79`	
`"Banana"`	`2.49`	✓
`"Ginger"`	`2.69`	

Next, we update the price of milk to $2.09. Milk is not currently in the cache, so we add it.

LRU Cache (Capacity 4)

Item	Price	LRU?
`"Apple"`	`3.79`	
`"Banana"`	`2.49`	✓
`"Ginger"`	`2.69`	
`"Milk"`	`2.09`	

Next, we update the price of tofu to $3.79. Tofu is not in the cache, so we need to add it. However, our cache is already filled to capacity, so we will need to evict the least recently used element, which is `"Banana"`. After `"Banana"` is evicted, we add in `"Tofu"`. Note that the least recently used element now becomes `"Ginger"`.

LRU Cache (Capacity 4)

Item	Price	LRU?
`"Apple"`	`3.79`	
`"Tofu"`	`3.79`	
`"Ginger"`	`2.69`	✓
`"Milk"`	`2.09`	

Next, we update the price of ginger to $2.79. Ginger is in the cache, so we update its value. The LRU element now becomes `"Apple"`.

LRU Cache (Capacity 4)

Item	Price	LRU?
"Apple"	3.79	✓
"Tofu"	3.79	
"Ginger"	2.79	
"Milk"	2.09	

Next, we update the price of banana to $2.59. Banana is not in the cache, so we evict the LRU of `"Apple"` and add `"Banana"` back in.

LRU Cache (Capacity 4)

Item	Price	LRU?
"Banana"	2.59	
"Tofu"	3.79	
"Ginger"	2.79	
"Milk"	2.09	✓

An outline of an `LRUCache` class for the above cache is shown below:

```
1   class LRUCache {
2   private:
3     size_t capacity;
4     // TODO: Add any data structures here!
5   public:
6     LRUCache(size_t capacity_in) : capacity{capacity_in} {}
7
8     // Gets the value of the key if it exists in the cache
9     // Time complexity: O(1) on average
10    std::optional<double> get(const std::string& key) {
11      // TODO: Implement code here
12    } // get()
13
14    // Updates the value of the key, inserting it if necessary
15    // If the cache is already at capacity, evict the least-recently used key
16    // Time complexity: O(1) on average
17    void set(const std::string& key, double value) {
18      // TODO: Implement code here
19    } // set()
20  };
```

Your goal is to implement the following two methods:

- `get()`: Returns the value of the given key if it exists in the cache; otherwise, return `std::nullopt`.
- `set()`: Updates the value of the given key to the given value if the key exists in the cache. Otherwise, add the key-value pair to the cache, evicting the least recently used key if the insertion would cause the cache to go over capacity.

Both `get()` and `set()` should run in $\Theta(1)$ time on average. For simplicity, assume the key is a `std::string` and the value is a `double` (like with the shopping items example above).

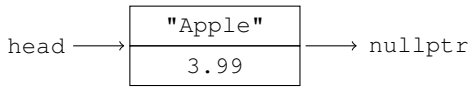
Since we want to store our data in a way that supports efficient key-value lookup, our solution should utilize an `std::unordered_map<>` that maps each key to its corresponding value. However, the tricky part of this problem is to devise a method that allows you to evict the least recently used key from the cache. A potential naïve solution would be to associate a "timestamp" with each key to identify when it was most recently used. From an efficiently standpoint, though, this approach is not ideal, since this would require us to complete a linear pass over all the timestamps to identify which item was least recently used (as well as to update the least recently used item after an eviction).

Another solution would be to maintain a `std::priority_queue<>` of key-value pairs with priority based on a timestamp. However, the use of a priority queue would make `get()` and `set()` run in $\Theta(\log(n))$ time. Is there another data structure that allows us to evict keys from our cache in constant time? The answer turns out to be a *doubly-linked list*. A doubly-linked list is useful here for two reasons:

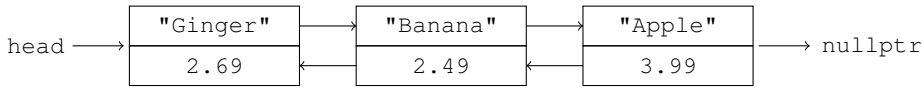
1. If you have access to a node of a doubly-linked list, you can remove it in *constant* time.
2. Lists do not have to worry about reallocation, and a node in a list is not invalidated until the node itself is removed.

When an item is inserted into the cache, we can insert it to the front of the list in constant time. Additionally, when an item is accessed using `get()` or modified using `set()`, we can also move it to the front of the list in constant time. This causes the least recently used elements to move toward the back of the list, allowing us to easily identify which keys should be evicted when the cache becomes full.

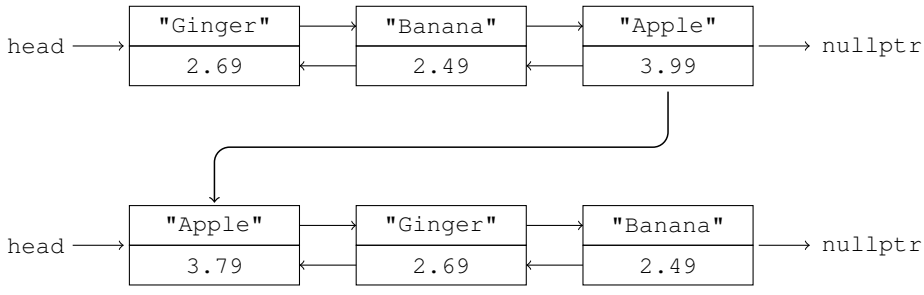
Let's look at how a doubly-linked list can be used to solve this problem using the previous example. We first insert the key-value pair of `{"Apple", 3.99}` to the front of our list, as shown.



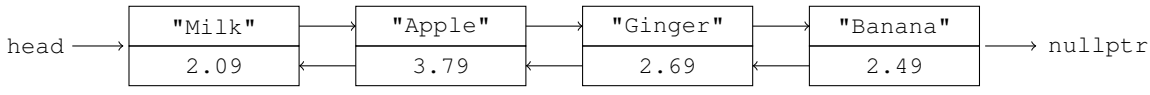
Next, we insert the key-value pairs of `{"Banana", 2.49}` and `{"Ginger", 2.69}` to the front of the list.



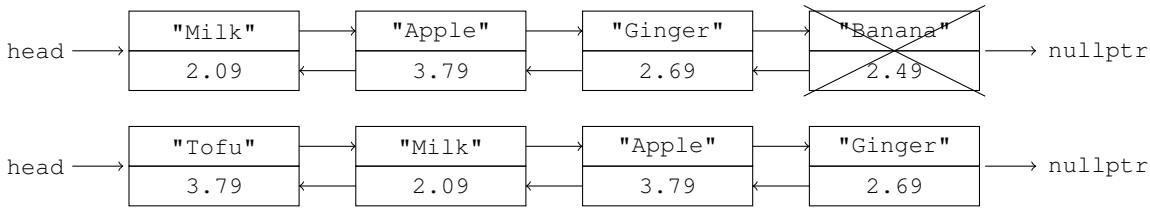
Next, we update the value of `"Apple"`. Since `"Apple"` was accessed, we move it to the front of the list, which can be done in constant time if we have access to the node of `"Apple"` (we will go over how to access this node in constant time later, but it has to do with the hash table we mentioned earlier). Notice that the item at the back of our list, `"Banana"`, is now our new least recently used element.



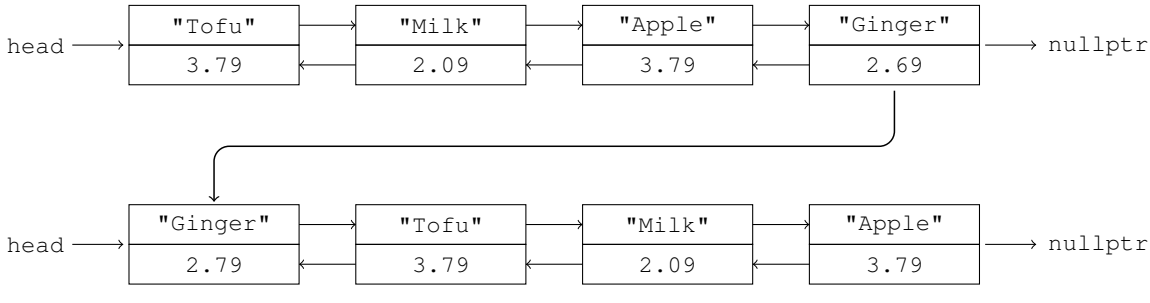
Next, we insert the key-value pair of `{"Milk", 2.09}` to the front of the list.



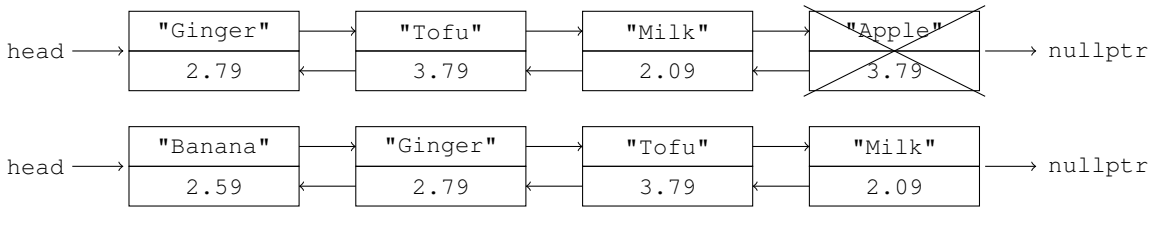
Next, we insert the key-value pair of `{"Tofu", 3.79}`. However, our cache is already at its capacity of 4, so we cannot add this new element without removing something in the cache first. Since the least recently used element is at the back of the list, we will remove `"Banana"` from the cache before adding in `"Tofu"`.



Next, we update the value of `"Ginger"`. Since `"Ginger"` was accessed, we move it to the front of the list, which leaves `"Apple"` as the current least recently used element.

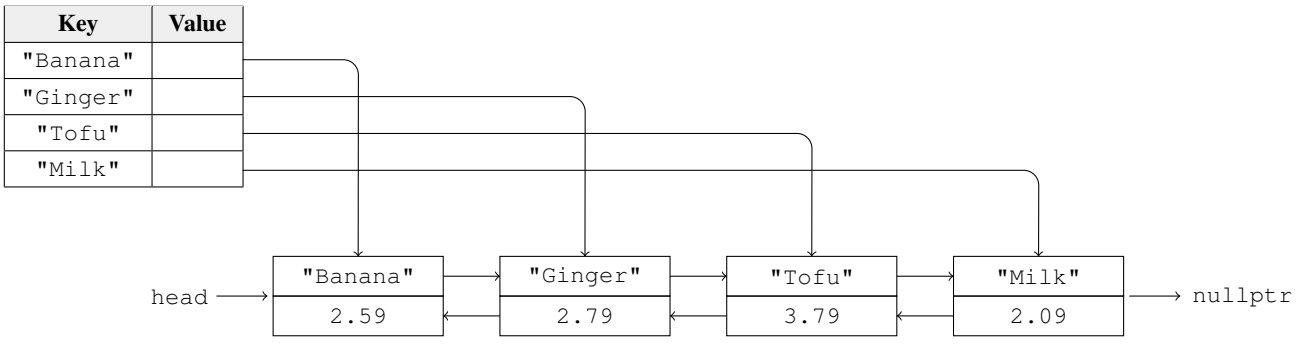


Next, we add `"Banana"` back into the cache. Since our cache is full, we will evict the least recently used item — which is `"Apple"` since it is at the back of the list — before adding in `"Banana"`.

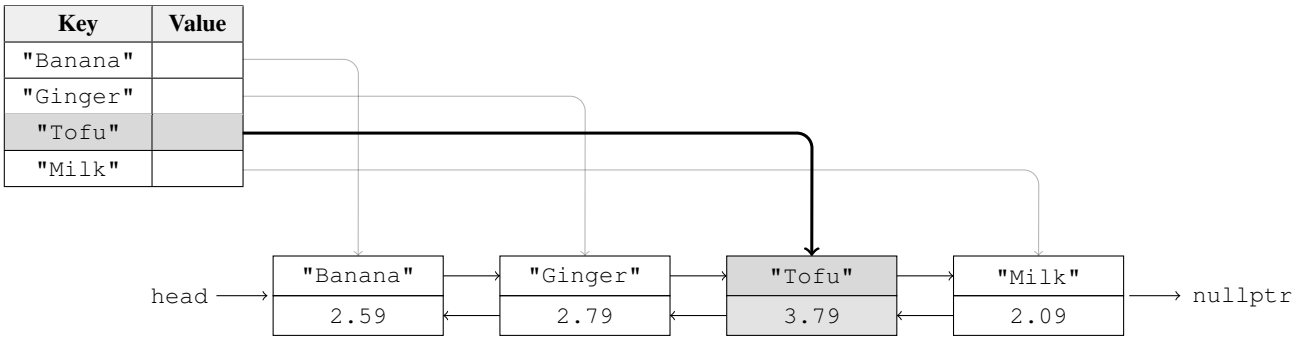


To perform this procedure efficiently, we would need a way to access the elements in the list in constant time on average (i.e., given the string `"Tofu"`, we need to be able to return the node corresponding to `"Tofu"`). However, if we just had a list on its own, we would need to iterate over the list to find the node associated with any given key. How can we support constant time access into the nodes of the list?

The solution is to *supplement the list with a hash table* that maps each item to an iterator that points to its node in the list (this is okay since iterators in a list are not invalidated until their corresponding items are removed). An illustration is shown below:



For example, if you wanted to update the value of `"Tofu"`, you can use the hash table to look up its node in the list in constant time.



A solution to the problem is implemented below. A `std::list<>` is used to store the items in the cache, and an `std::unordered_map<>` is used to map items to their corresponding node in the list. Whenever an item is updated or added, it is moved to the front of the list; whenever an item needs to be evicted from the cache, it is removed from the back of the list. Because unordered maps support $\Theta(1)$ time lookup on average, and lists support $\Theta(1)$ time insertions and deletions when given a node, the overall time complexities of `get()` and `set()` are $\Theta(1)$.

```
class LRUCache {
private:
  size_t capacity;
  std::list<std::pair<std::string, double>> cache_list;
  std::unordered_map<std::string, std::list<std::pair<std::string, double>>::iterator> cache_map;
public:
  LRUCache(size_t capacity_in) : capacity{capacity_in} {}

  std::optional<double> get(const std::string& key) {
    auto it = cache_map.find(key);
    if (it == cache_map.end()) {
      return std::nullopt;
    } // if
    // the std::list::splice() method can be used to move the accessed node to the the front
    // of the list, but you can also do a standard erase() and push_front() instead
    cache_list.splice(cache_list.begin(), cache_list, it->second);
    return it->second->second;
  } // get()

  void set(const std::string& key, double value) {
    auto it = cache_map.find(key);
    if (it != cache_map.end()) {
      cache_list.splice(cache_list.begin(), cache_list, it->second);
      it->second->second = value;
    } // if
    else {
      // if cache is at capacity, evict element at back of list and remove it from map
      if (cache_map.size() == capacity) {
        std::string item_to_evict = cache_list.back().first;
        cache_list.pop_back();
        cache_map.erase(item_to_evict);
      } // if
      cache_list.emplace_front(key, value);
      cache_map[key] = cache_list.begin();
    } // else
  } // set()
};
```

Example 17.10 You are given two vectors containing strings. Write a function that returns the strings that *are in both vectors* with the *smallest index sum*. The index sum of a string that exists at index `i` of the first vector and index `j` of the second vector is `i + j`. You may return the strings in any order. There are no duplicates in each of the vectors, and the vectors are small enough that you do not need to worry about integer overflow when calculating the index sum. The function declaration is as follows:

```
std::vector<std::string> minimum_index_sum(const std::vector<std::string>& vec1,
                                           const std::vector<std::string>& vec2);
```

Example: Given the following two vectors:

```
vec1 = ["Nikhil", "Daniel", "Aray", "Reyna", "Brian", "James", "Gavin"]
vec2 = ["Gavin", "Mert", "Reyna", "Brian", "Daniel", "Zach", "Aasher", "Nikhil"]
```

Of the strings that exist in both vectors, these are their index sums:

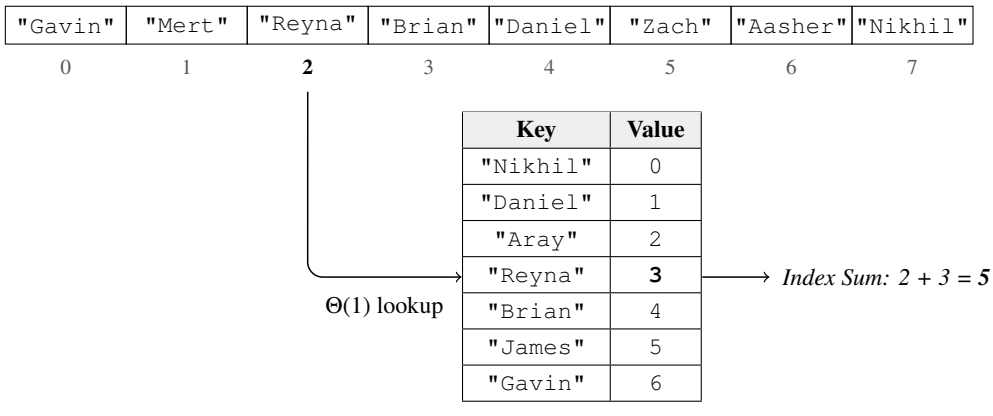
String	Index in `vec1`	Index in `vec2`	Index Sum
`"Brian"`	4	3	`4 + 3 = 7`
`"Daniel"`	1	4	`1 + 4 =` **5**
`"Gavin"`	6	0	`6 + 0 = 6`
`"Nikhil"`	0	7	`0 + 7 = 7`
`"Reyna"`	3	2	`3 + 2 =` **5**

The smallest index sum among the strings is 5, so the function should return the vector `["Daniel", "Reyna"]` (in any order).

A naïve solution would be to iterate over the two vectors using a nested `for` loop. For each element we visit in the outer loop, we iterate over the other vector to identify the index sum of that element if it exists. Along the way, we keep track of the best solution(s) we have encountered so far, updating it whenever we encounter an element with a better index sum. An implementation of this naïve solution is shown below:

```
1  std::vector<std::string> minimum_index_sum(const std::vector<std::string>& vec1,
2                                             const std::vector<std::string>& vec2) {
3    std::vector<std::string> solution;
4    size_t best_idx_sum = std::numeric_limits<size_t>::max();
5
6    for (size_t idx1 = 0; idx1 < vec1.size(); ++idx1) {
7      std::string& curr = vec1[idx1];
8      for (size_t idx2 = 0; idx2 < vec2.size(); ++idx2) {
9        if (vec2[idx2] == curr) {
10         size_t idx_sum = idx1 + idx2;
11         if (idx_sum < best_idx_sum) {
12           solution.clear();
13           solution.push_back(curr);
14           best_idx_sum = idx_sum;
15         } // if
16         else if (idx_sum == best_idx_sum) {
17           solution.push_back(curr);
18         } // else if
19       } // if
20     } // for idx2
21   } // for idx1
22
23   return solution;
24 } // minimum_index_sum()
```

The time complexity of this solution is $\Theta(mn)$, where m and n are the lengths of the two vectors. How can we improve this solution? Notice that the inefficiency is caused by the nested loop, since we have to repeatedly iterate over one of the vectors once for each element in the other vector. However, the only reason we are performing this nested loop is to complete a lookup: given a string in one vector, we want to find its index (if it exists) in the other vector. Therefore, we can circumvent the need for a nested loop by inserting the contents of one vector into an `std::unordered_map<>` that maps each element to its index. Then, when iterating over the other list, we can query the hash table in *constant* time to find the index of an element, instead of having to perform a linear time traversal.



This improved solution is implemented below:

```
std::vector<std::string> minimum_index_sum(const std::vector<std::string>& vec1,
                                           const std::vector<std::string>& vec2) {
  std::vector<std::string> solution;
  size_t best_idx_sum = std::numeric_limits<size_t>::max();

  // insert all the strings in one vector into a hash table that maps string -> index
  std::unordered_map<std::string, size_t> idx_map;
  for (size_t idx1 = 0; idx1 < vec1.size(); ++idx1) {
    idx_map.emplace(vec1[idx1], idx1);
  } // for idx1

  // iterate over the other vector, querying the hash table to identify index in first vector
  for (size_t idx2 = 0; idx2 < vec2.size(); ++idx2) {
    std::string& curr = vec2[idx2];
    auto it = idx_map.find(curr);
    if (it != idx_map.end()) {
      size_t idx_sum = it->second + idx2;
      if (idx_sum < best_idx_sum) {
        solution.clear();
        solution.push_back(curr);
        best_idx_sum = idx_sum;
      } // if
      else if (idx_sum == best_idx_sum) {
        solution.push_back(curr);
      } // else if
    } // if
  } // for idx2

  return solution;
} // minimum_index_sum()
```

Since our loops are no longer nested and are instead performed sequentially, the time complexity now becomes $\Theta(m+n)$, where m and n are the lengths of the two vectors. This is because the bodies of both loops can be completed in average-case constant time (since insertion and lookup in a hash table can both be done in $\Theta(1)$ time on average). Notice that an additional optimization we can make is to add the *smaller* of the two vectors to the unordered map — this allows us to reduce our memory usage, as we only need to store the contents of one of the vectors.