Informatics 2: Introduction to Algorithms and Data Structures
Notes on Lecture 8
Sets, dictionaries and hashing

John Longley
© University of Edinburgh 2026

In Lecture 7, we looked at three classic data structures—lists, stacks and queues—considering both the abstract interfaces that specify the operations on these structures and a number of concrete implementations of them. In this lecture, we'll carry out a similar programme for two more classic datatypes—*sets* and *dictionaries*. Again, we'll start with some abstract interfaces for these types, but most of the lecture will be devoted to an implementation technique known as *hash tables*, which often turns out in practice to be an extremely fast way of supporting them.

Sets and dictionaries

What do we mean by a *set*? Loosely, a set is just any collection of items, not thought of as presented in any specific order. For instance, in mathematical notation we might write $\{3, 5\}$ for the set containing just the integers 3 and 5 and nothing else, and this very same set can also be written as $\{5, 3\}$. Here we refer to 3 and 5 as the *members* or *elements* of the set. In mathematics, sets may be either finite or infinite, but here we'll be using the term *set* to mean 'finite set'. We'll be thinking about sets whose members are all of some given type X.

What operations do we want to be able to perform on a set? Most importantly, we want to be able to test whether or not a given item $x : X$ is a member of the set: that is, whether the set *contains* that item. We would also like operations for inserting a new element into a set, deleting an existing element, and testing whether a set is empty.[1] We can think of each of these operations as a 'method' associated with a given set, and the following abstract interface for sets gives their expected types.

$$\begin{aligned} \textbf{contains} &: X \to \texttt{bool} \\ \textbf{insert} &: X \to \texttt{void} \\ \textbf{delete} &: X \to \texttt{void} \\ \textbf{isEmpty} &: \texttt{void} \to \texttt{bool} \end{aligned}$$

[1] We won't worry here about what happens if we try to insert an item that's already there, or delete one that's not there. Our policy in these cases makes no real difference, given that we can always test in advance whether the item in question is present.

© *University of Edinburgh 2026*

What do we mean by a *dictionary*? A dictionary resembles a set in that it contains finitely many entries of some type X; but now each of these entries has some associated information of some type Y, so that a typical element of the dictionary will have the form (x, y) where $x \in X, y \in Y$. So we can think of a dictionary as basically a table in which we can look up the information associated with a given $x : X$, and if the dictionary contains some entry (x, y), we'll get back the value y. (For example, in an English dictionary, the keys might be English words and the values might be their associated definitions.) Here the elements of type X are referred to as *keys*, and those of type Y are referred to as *values*. It is of course essential that any dictionary will associate at most one value y to a given key x.

We might also want to be able to insert a new key-value pair into a dictionary, and to delete the entry for a given key x if there is one. For good measure, we can also add an operation for testing whether a given key is present in the dictionary, and one for testing whether a dictionary is empty or not. We thus have the following abstract interface for dictionaries:

contains	$: X \to$ `bool`
lookup	$: X \to Y$
insert	$: X * Y \to$ `void`
delete	$: X \to$ `void`
isEmpty	: `void` $\to$ `bool`

Both sets and dictionaries are supported in Python, which has some special syntax for both of them. Here are some little examples, illustrating what the **contains** operation for sets and the **lookup** operation for dictionaries look like:

```
Beatles = {'John', 'Paul', 'George', 'Ringo'}
'George' in Beatles               # returns True

BeatlesYearsOfBirth =
    {'John':1940, 'Paul':1942, 'George':1943, 'Ringo':1940}
BeatlesYearsOfBirth['George']     # returns 1943
```

Some naive implementations

How might sets and dictionaries be implemented? Since we already know how to implement lists in several ways, a very first step might be simply to store them simply of lists of items or of key-value pairs. For example, we could in principle imagine the Python set and dictionary just mentioned being *represented* under the hood by the following Python lists:

```
Beatles_Rep = ['John', 'Paul', 'George', 'Ringo']
BeatlesYearsOfBirth_Rep =
    [('John',1940), ('Paul',1942), ('George',1943), ('Ringo',1940)]
```

But would this be a *good* way to represent sets and dictionaries? In general, it's a terrible way, because it means that if we want to search for a given item or key, we have no choice but to search through the list until we find it (or reach the end of the list without finding it). For a list of length n, this would of course take $\Theta(n)$ time in the worst case. In fact, even in the average case, a successful

lookup will take $\Theta(n)$ time, since on average we can expect to have to look at $n/2$ of the list elements. This is very bad if the main thing we want a set or dictionary for is to be able to perform **contains** or **lookup** efficiently.

What if our lists are *sorted* by key (say alphabetically if our items or keys happen to be strings)? This certainly makes **contains** and **lookup** much easier, because we can now do them very efficiently by *binary search*. Broadly, if we're searching for a key x, we start by comparing x with a key in the middle of our list, and this will tell us whether x (if present at all) will be in the first or second half of the list. If it's in the first half, we then compare x with a key a quarter of the way through the list, and this will tell us whether we need to search the first or second quarter. We continue in this way, narrowing down the portion of the array we need to search until we get down to a single entry. We then just have to test whether the key of this entry is in fact x. The following simple pseudocode embodies this idea (minor variations and improvements are possible):

```
binarySearch'(A,key,i,j):      # searches A[i], ..., A[j−1]
    if j−1 = i
        if A[i].key = key then return A[i].value else FAIL
    else
        k = ⌊ i+j/2 ⌋
        if key < A[k].key then return binarySearch'(A,key,i,k)
        else return binarySearch'(A,key,k,j)

binarySearch(A,key):       return binarySearch'(A,key,0,|A|)
```

This is a recursive algorithm in which the length of the list segment we're searching (namely sf j-i) gets halved with each recursive call. It's therefore intuitively clear that for a list of length n, the number of such recursive calls will be $O(\lg n)$, and indeed (assuming we're using an array implementation of lists) that the total runtime will be $O(\lg n)$. So this is a massive improvement over unsorted lists: even if a dictionary has a million entries, it won't take more than 20 steps to locate the entry we want. This is in essence why the entries in an old-fashioned physical dictionary are listed in alphabetical order.

However, this isn't the whole story, because even sorted lists won't be great for doing inserts or deletes. For the list implementations we've considered so far, these operations will still have $\Theta(n)$ runtime in the worst and even the average case, for instance because they will involve shifting a lot of elements by one position in the carrier array. So it's reasonable to ask: can we do better? Is there some other way of implementing sets or dictionaries that gives good performance for all the operations we've mentioned, at least most of the time? This is what is achieved by the technique of *hash tables*.

Hash tables: The key idea

What's the main idea behind a hash table? To explain this, we'll start with an idea that as it stands is really silly, then see how it can be modified to be not quite so silly.

Suppose we want to store a dictionary with strings as keys, and let's say there's an upper limit of 40 on the length of these strings. The really silly idea

is this. First, fix on some way of translating every string s to an integer $\imath(s)$, in such a way that no two strings get translated to the same integer. This isn't hard to do: for instance, if our strings consist of ASCII characters, each of which has a numerical code from 0 to 127, then we can simply view our string as a number written in base 128. For instance, the characters `c`, `a`, `t` respectively have ASCII codes 99, 97, 116, so we could translate the string `'cat'` to the integer

$$(99 * 128^2) + (97 * 128) + 116 = 1634548 ,$$

from which we could recover the string `'cat'` by some boring calculation. The idea now is to try to store a dictionary as a massive array D, in which the value associated with a key s is stored in $D[\imath(s)]$. Thus, for the above example, the value associate with `'cat'` (if there is one) would be stored in $D[1634548]$.

Why won't this idea fly? Think how large the array D would need to be. If a string s can be any sequence of ASCII characters of length up to 40, the possible values of $\imath(s)$ will range from 0 to $128^40 - 1$, so this is how large our array would need to be, given that we don't know in advance which strings will actually feature as keys in our dictionary. This is obviously infeasibly large—and even if we could store an array this big, almost all of it will remain unused for a realistic dictionary that has perhaps a million or a billion genuine entries.

But there's something we can salvage from this silly idea. Suppose that instead of trying to convert strings to integers in a way that preserves all information about the strings, we use some function $\sharp$, called a *hash function*, which maps our large set of potential keys down to a much smaller set of integers, let's say the integers $0, \ldots, m-1$ for some reasonable m. (We have in mind here that m is broadly of the same order of magnitude as the number n of dictionary entries we want to store—maybe larger, maybe smaller, but in roughly the same ballpark.) We'll refer to these integers $0, \ldots, m-1$ as our *hash codes*. A very simple example of such a hash function might be:

$$\sharp(s) = \imath(s) \mod m ,$$

where $\imath$ is the mapping from strings to integers already defined above. There are many other ways in which we might define a hash function, but however we do it, we should ensure that it's fast and easy to compute. The idea, now, is to store our dictionary using an array D of size m, using the cell $D[\sharp(s)]$ to store the value associated with the key s.

Why is this called *hashing*? The idea is that when we apply our hash function $\sharp$ to a key s, we destroy or lose most of the information about s: only a small remnant of information is left in $\sharp(s)$. And ideally, if our hash function is well chosen, even what information does remain is all scrambled up by the hashing process, so that it's not easy to tell anything of significance about s just from knowing $\sharp(s)$. This helps to ensure that $\sharp$ distributes strings evenly among the m possible hash codes, which as we will soon see is a desirable property.

This, at least, is the first stage in how hash tables work: we 'hash down' our infeasibly large set of potential keys to a manageable set of hash codes.

The problem of collisions

The basic problem with this scheme is very obvious: since there are far more potential keys s than hash codes, there will be many pairs of keys s, s' such that

$\sharp(s) = \sharp(s')$. What if we want to use both s and s' as keys in our dictionary, so that there are two entries competing for the same slot in our array D? In this situation, we say that we have a clash, or *collision*, between the keys s, s'. This problem of collision is the basic challenge that a hash table will need to overcome, and we'll look at three possible responses to it.

Before we do so, let's ask how likely it is that such collisions will occur. Could the problem perhaps be avoided in practice simply by taking m comfortably larger than our expected number of entries n, so the hash codes in use are only sparsely distributed among the numbers $0, \ldots, m-1$?

There's an appealing analogy which helps to illustrate what's going on. Suppose our 'keys' are actual people, and consider the hash function $\sharp$ that maps each person p to the birthday of p—or more precisely, to a number in the range $0, \ldots, 365$ representing their birthday. Obviously, if we have a set of 400 people, it's inevitable that some will share a birthday—but perhaps, with significantly fewer than 366 people, such collisions become unlikely? More specifically, we can ask: how many people must we have (in the same room, say) for it to be more likely than not that there are two who share a birthday?

One common intuitive response to this is to suppose that this threshold occurs at somewhere around 180 people—half the number of potential birthdays. In fact, the answer is surprisingly small: once we have 23 people, the probability of a shared birthday becomes > 0.5: a fact sometimes known as the *Birthday Paradox*.[2] This may seem counterintuitive, because with only 23 people, the vast majority of days of the year won't be anyone's birthday. But perhaps a way to think about it is to consider the number of *pairs* of people who might be sharing a birthday. In the case of 23 people, many of you will know from DMP that the number of (unordered) pairs will be $23 \times 22/2 = 253$, which is of broadly the same order of magnitude as 366. Of course, this is only a loose intuition—if you're curious to see the mathematical proof of the Birthday Paradox, details are given in the CLRS textbook (Section 5.4.1).

One can also ask about the probability of more than two people sharing a birthday. In the lecture slides, I gave approximate probabilities that in a class of 347 students there would be some birthday shared by 2, 3, 4, 5 or 6 people. Perhaps surprisingly, it turns out that in this case the probability of 5 people sharing a birthday is still around 66%.

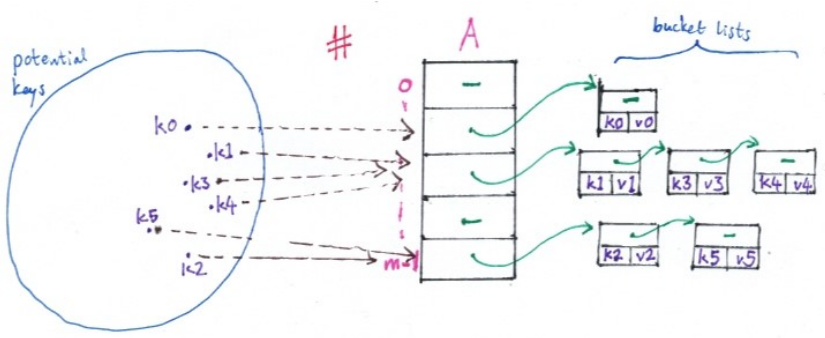
Returning now to hash functions, this shows that even our hash table size m is significantly larger than the number n of keys actually in use, it's still overwhelmingly probable that collisions *will* occur—and we can even expect a few hash codes that are shared by more than 2 keys. So we have to accept that *collisions are a fact of life*, and we'll need to find a way of dealing with them.

Solution 1: Bucket lists

Our first solution is as follows. We simply accept that rather than storing just a single item or dictionary entry for each hash code, we may in general have

[2] This is based on the assumption of a uniform distribution: in other words, we're supposing that people's birthdays are evenly spread over all days of the year. That of course isn't true in practice—not just because of February 29, but also because it's well known that some months of the year are more popular for being born in than others. But in fact, it turns out that if there's a non-uniform distribution, then that only further increases the probability of shared birthdays.

to store a *list* of items. We refer to those lists *bucket lists*, the idea being that keys that hash to the same code get put into the same 'bucket'. Because these bucket lists will be of greatly varying sizes (many are likely to be empty), but probably very few of them will be long and none of them will be *too* long, linked lists are a good choice here for how to represent them. Thus, we have an array A of size m with a cell for each hash code, and what we store in cell $A[i]$ will be not a dictionary entry itself, but a reference to the head of the linked list for the relevant bucket. (Or a null reference, if this bucket is currently empty). In the case of a dictionary, each element of these linked lists will store a single dictionary entry, i.e. a key-value pair. This is illustrated by the following picture, which suggests the idea of a vast universe of *potential* keys k, of which only a small proportion (written as $k0, \ldots, k5$) are actually in use.



If it were a set rather than a dictionary we were storing, we'd simply omit the v components.

How do the operations on this dictionary implementation work? To look up the value (if any) associated with a key k, we first compute the hash code $i = \sharp(k)$, then search through the linked list referenced by $A[i]$ to see if there's an entry (kj, vj) where $kj = k$. Likewise, to insert a new key-value pair (k, v), we might first do the above to check if an entry for k is already present—assuming it isn't, we then create a new cell for (k, v) and add it to the linked list at $A[i]$ (e.g. at the start of the linked list). To delete an entry with key k, we again locate the linked list cell for k (if there is one) and remove it from the linked list by making the cell immediately before it point to the one immediately after. The idea is that since in practice almost all of the bucket lists will be short, all these operations should work efficiently in practice.

Can we quantify this somehow? Suppose we have a dictionary storing n entries—so that n is the number of keys actually in use—and we're using an array of size m, where m is the number of possible hash codes. An important statistic for our hash table will be the ratio $\alpha = n/m$, known as the *load factor* for the table. Because the total number of cells in all buckets will be n, it's clear that this α will be the *average* size of the buckets (regarding empty buckets as having size 0). Supposing that we know the approximate size n of the dictionary we're wanting to store, we are basically free to choose what we'd like this load factor to be. To achieve a load factor $\alpha < 1$, we would choose $m > n$, which means there will inevitably be some empty buckets—and the smaller our α, the sparser our table will be, so the smaller the average bucket size and the faster our dictionary operations will be—but the more space our array will take up

in memory because m will be large. Likewise, if we opt for a large load factor $\alpha > 1$, we will save some space in our array, but our operations will tend to be slower because the bucket lists on average will be longer. So in this sense, the choice of α embodies the *time vs. space tradeoff* for our hash table.[3]

We can make this a bit more precise as follows. Let's continue to use n and m as above, and also write b_i for the number of entries in the bucket for hash code i, for each $i < m$. We are interested in the *average case* runtime for a lookup on key k. Recall that a dictionary lookup may be be either *successful*, if we find an entry that matches k, or *unsuccessful*, if k isn't present in the dictionary. It turns our that the runtime analysis is slightly different in these two cases. Here we'll just consider the case of unsuccessful lookups, which is the easier of the two.

Suppose, then, that we're trying to lookup a key k that isn't actually present, and let's make the idealizing assumption that such a k is equally likely to hash to any one of our m possible hash codes. (Part of what we want from a well-designed hash function is to help us to achieve such a uniform distribution in practice.) Writing $i = \sharp(k)$, we have to search the bucket list for i to see if any of its elements match k—and because none of them do, this will mean we do exactly b_i equality comparisons between k and keys appearing in the bucket list. So what's the *average* number of key comparisions? If all m hash codes are equally likely, we're just taking the average over all hash codes i of the number b_i: that is,

$$(\sum_{i=0}^{m-1} b_i)/m \;=\; n/m \;=\; \alpha\,.$$

So the average number of key comparisons will be none other than our load factor α. We should also of course account for the time taken to compute $\sharp(k)$ from k; but this will only be done once per lookup, and if we assume that $\sharp$ can be computed within constant time, this won't affect the asymptotics and we can summarize the situation by saying that the average time for an unsuccessful lookup is $\Theta(\alpha)$.[4]

In CLRS Section 11.2, a corresponding analysis is done for the case of a successful lookup, showing that this too has average runtime $\Theta(\alpha)$ under the assumption that all keys appearing in the table are equally likely to be looked up. (This is not assessable course material.)

Just to recall here what we've learned from the Birthday Paradox: although α gives the *average* number of key comparisons, because of the unevenness of the bucket sizes, it's extremely likely that occasional lookups will require significantly more comparisons. For instance, drawing on our earlier example, if we were storing records for 346 using a hash table of size 366, the load factor

[3]Having decided on roughly what we'd like our α to be, and chosen a suitable array size m, the question arises as to what happens if our dictionary becomes so large that the load factor is higher than we'd like. A perfectly reasonable policy here would be to do what we did for extensible arrays: we can choose a larger value of m and a new hash function $\sharp$ to go with it, then expand our array to one of the new size, copying all the existing entries to the new table which will involve rehashing all the existing keys with the new hash function. This may seem an expensive thing to do, but just as with extensible arrays, it turns out that the amortized cost per insert operation can be quite reasonable.

[4]This makes formal sense if we consider some limiting process in which α tends to infinity. This might seem a slightly odd thing to consider—but nonetheless, we can take away the broad idea that the average lookup time will grow linearly with the load factor.

α would be less than 1, but it's highly likely that there will be some hash codes i whose buckets contain 4 or even 5 items, and for keys that hash to such an i, the execution time of our dictionary operations will be correspondingly larger.

Making a *proper* hash of it

Let's now reflect a little more on how we might choose our hash function $\sharp$, for which we've given only one very simple example so far. The two main properties we'd like are as follows:

1. $\sharp$ should be easy and fast to compute.
2. $\sharp$ should be such that, as far as possible, all possible hash codes $i < m$ occur with equal probability as values for $\sharp(k)$, to help ensure that our table entries are spread as evenly as possible among the buckets.

The challenge in achieving 2 is that it's rarely the case in practice that all potential *keys* are equally probable. For instance, suppose our potential keys are strings that represent people's names. Obviously, in practice, not all strings of a given length are equally likely to be someone's name, and not all ASCII characters will be equally common. So we'd like, if possible, for our $\sharp$ to scramble things up so well that any distributional bias in the set of keys is lost when we pass to their hash codes. How might we achieve this?

Earlier, we suggested the simple idea of treating a string s as representing a number $\imath(s)$ in base 128, then taking $\sharp(s) = \imath(s) \mod m$ for some m. However, whether this works well or not will depend a lot on our choice of m. For example, a particularly *bad* choice for m would be any power of 2, such as 128 itself. This is because if s is the base 128 representation of $\imath(s)$, then $\imath(s) \mod 128$ will be simply the ASCII code for the last character of s—and this certainly won't give us a nice uniform spread of hash codes, because for instance the last character of someone's name is far more likely to be `e` than `(`. Even if we use a larger power of 2 like $2^14 = 16384$, we're still only picking up the last two characters of s, and there will be very significant distributional bias here.

Another bad choice would be 127. Why? Recall from Lecture 2 that if we write a number n in base 10, then $n \mod 9$ is determined by the sum of the digits. Transposing this idea to base 128, we see that $\imath(s) \mod 127$ is determined by the sum of (the ASCII codes for) the characters of s. This means, among other things, that two strings s, s' that are *anagrams* of each other, such as `algorithm` and `logarithm`, will hash to the same code. At least in some contexts, it turns out that such phenomena lead to a noticeable imbalance in our spread of hash codes.

A much better choice would be to take m to be some prime number that's not too close to a power of 2, such as 97: this will scramble things up quite nicely for us. On the other hand, $\imath(s) \mod 97$ is perhaps a little expensive to compute, so one might wonder whether we can do better. This leads us into the subtle and delicate art of hash function design, which is quite a substantial area which we won't explore further here.[5] The important point is simply to note that achieving both goals 1 and 2 above in a satisfactory way is not entirely trivial, but that a number of good solutions to this problem have been developed.

[5] For a sample of approaches, see the second half of the Wikipedia page for 'Hash function'.

Whatever we do, however, the *worst case* performance for these kinds of hash tables will always be terrible. The theoretical worst case will be when all n of our keys hash to the same code i, so that all our entries are stored in a single bucket list of length n. Clearly, this means that lookup and other operations will have worst case runtime $\Theta(n)$. Of course, under normal usage, this will be absurdly unlikely to happen if our hash function is well-chosen. However, if we were running a database server using such a hash table, we might be concerned about the possibility that a malicious user who knew our hash function could deliberately flood the database with keys all hashing to the same code, which would cause our operations to be painfully slow whenever that hash code came up. There are various ways to guard against this—e.g. by introducing an element of randomization so that the attacker won't know exactly what our hash function is, or else by using a 'cryptographic' hash function with the property that it is computationally hard to find many keys that hash to the same code—but we won't pursue these lines of thought further here.

Solution 2: Hash-probe tables

We now turn to a radically different, and quite ingenious, solution to the problem of collisions. Again, we use an array A of size m, where m is the number of hash codes, and refer to A as our 'hash table'. However, in this scheme, the dictionary items—the key-value pairs (k, v)—will live within the cells of A itself, rather than in linked lists outside of A. So how do we deal with collisions? In effect, we use a hashing scheme for which each potential key k has not just a single hash code $\sharp(k)$, but a whole choice of hash codes. More specifically, we give a way to compute a hash code $\sharp(k, i)$ where i can be any number from 0 to $m - 1$. The idea, broadly, is that for any key k,

- $\sharp(k, 0)$ will be our *first choice* of hash code for k: the one we'd prefer to use if it is available.
- $\sharp(k, 1)$ will be our *second choice* of hash code, in case $\sharp(k, 0)$ is not available.
- $\sharp(k, 2)$ will be our third choice, in case both $\sharp(k, 0)$ and $\sharp(k, 1)$ are already taken; and so on.

In this way, we get a whole sequence $\sharp(k, 0), \sharp(k, 1), \sharp(k, 2), \ldots$ of possible hash codes for k in order of preference.

Obviously, for a given k, we'd like all these hash codes $\sharp(k, i)$ to be distinct: there is no point in having $\sharp(k, 5) = \sharp(k, 3)$ since $\sharp(k, 5)$ is intended to be a code we might consider only when $\sharp(k, 3)$ is unavailable. So if we have a code $\sharp(k, i)$ for each $i < m$, it must therefore be that the whole sequence

$$\sharp(k, 0),\ \sharp(k, 1),\ \sharp(k, 2),\ \ldots,\ \sharp(k, m - 1)$$

being a sequence of m distinct hash codes each in the range $0, \ldots, m - 1$, must contain every possible hash code *once and only one*. In other words, this sequence will be a *permutation* of the list $0, \ldots, m - 1$. The point of this is that if *any* hash code c is still available, then c will eventually come up in our sequence of possible hashes for k: for some i, we'll have $\sharp(k, i) = c$. We refer to the above sequence $\sharp(k, 0), \ldots, \sharp(k, m-1)$ as the *probe sequence* for k, for reasons that will emerge shortly.

As we know, the number of possible permutations of $0, \ldots, m-1$, or probe sequences, will be $m!$. And the idea is that different keys k will give rise in the above way to different probe sequences. Indeed, the theoretically ideal situation would be for each of the $m!$ possible probe sequences to arise with equal probability as k varies over the set of keys. This will mean that our probe sequences in practice behave essentially randomly and unpredictably: for instance, knowing $\sharp(k, 0)$ won't enable us to predict anything at all about what $\sharp(k, 1)$ is likely to be (except that it must be different from $\sharp(k, 0)$), and so on. All of this will help to ensure a nice uniform spread of entries in our hash table without significant biases or bunching effects. In practice, this ideal situation of $m!$ equally probable probe sequences isn't so feasible to attain in practice, but with a suitable choice of $\sharp$ we can get close enough. (Once again, there's a whole delicate art of the design of such hash-probe functions which we won't go into here.)

How are these hash-probe functions used? We'll illustrate with some examples, starting with the **insert** operation. The idea is as follows: to insert an entry with key k into our table A, we simply 'probe' the cells $A[\sharp(k, 0)], A[\sharp(k, 1)], \ldots$ in turn until we find a cell that's currently free (i.e. not already occupied by another table entry), then insert our entry into this free cell. Recall that if there *is* a free slot anywhere in the table, it's guaranteed that our probe sequence will enventually find it. Of course, if there are only a few free slots, it might take a long time to find one; however, if say 20% of the array cells are free, then each probe will (ideally) have a 1 in 5 chance of finding a free slot, so we can expect that most of the time we'll find one after not too many probes, and the probability that you'll have to do more than, say, 30 will be $0.8^30 \approx 0.0012$.

For the sake of our example, we'll suppose for simplicity that we're just trying to store a set, nor a dictionary, so that entries will be just keys rather than key-value pairs. The set will be a set of natural numbers, and we'll use an array A of size $m = 10$ to store them. Our hash-probe function will be a very simple one, chosen to make the mental arithmetic easy:

$$\sharp(k, i) \;=\; (k + i) \mod 10 \,.$$

Notice that for any fixed k, the probe sequence $\sharp(k, 0), \sharp(k, 1), \ldots$ will cycle through the numbers $0, \ldots, 9$ in some order. We'll also suppose that our table A is initially empty—all slots are free—where we can record that an array cell is free e.g. by storing -1 in it.

Let us try three insertions in succession, starting from an empty table

- Let's first insert 49 into our set. Our first choice of hash code for 49 is $\sharp(49, 0) = 9$, and since $A[9]$ (like all cells of A) is currently free, we simply set $A[9] = 49$.
- Next, let insert 28. Once again, we calculate $\sharp(28, 0) = 8$, and since $A[8]$ is currently free, we set $A[8] = 28$.
- Now suppose we try to insert 58.
  - First, we calculate $\sharp(58, 0) = 8$. We look in $A[8]$, and find that this cell is already taken ($A[8] = 28$).
  - We therefore move on to our second choice: $\sharp(58, 1) = 9$. We look in $A[9]$, and see that this too is already taken ($A[9] = 49$).

– We now consider our third choice: $\sharp(58, 2) = 60 \mod 10 = 0$. Since $A[0]$ is currently free, we may insert 58 there.

The final state of our array after these three insertions will therefore be:

0	1	2	3	4	5	6	7	8	9
58	-1	-1	-1	-1	-1	-1	-1	28	49

Now let's consider how **contains**(k) will work. Once again, we successively probe the cells $A[\sharp(k, 0)], A[\sharp(k, 1)], \ldots$ until we either find a cell containing k or encounter an unoccupied cell. Let's illustrate this with three membership tests using the hash table shown above.

- First, let's test whether 28 is present. We start by calculating $\sharp(28, 0) = 8$, then probe $A[8]$ to see if it contains the key we're looking for. Since it does, we can immediately return *True*.

- Next, let's test if 58 is present. Again, we first look at $A[\sharp(58, 0)] = A[8] = 28$. Since $28 \neq 58$, this isn't what we're looking for, but perhaps 58 is present somewhere else in the array? So we carry on and look at $A[\sharp(58, 1)] = A[9] = 49$. Since $49 \neq 59$, we must still carry on. Now we look at $A[\sharp(58, 2)] = A[0] = 58$. This is the key we're looking for, so we can return *True*.

- Finally, let's test if 39 is present. We first try $A[\sharp(39, 0)] = A[9] = 49 \neq 39$. We then try $A[\sharp(39, 1)] = A[0] = 58 \neq 39$. We now try $A[\sharp(39, 2)] = A[1] = -1$. At this point, we can conclude that 39 doesn't appear anywhere in the table. This is because if 39 had been inserted at some point in the past, the probings for the insertion would have ended once we encountered the free slot aat $\sharp(39, 2)$—or perhaps earlier if the slot at $\sharp(39, 0)$ or $\sharp(39, 1)$ had been free at the time of the insertion. In any case, the probings for the insertion wouldn't have proceeded *beyond* $i = 2$ where there would have been a free slot. So we can confidently return *False*.

Notice that for this kind of hash table, the ratio $\alpha = n/m$ is always at most 1. The closer α is to 1, the more full the table will be, and so the longer we expect it to take to locate a free slot. Indeed, under various idealizing assumptions, it can be shown that the expect number of probes for an insertion or an unsuccessful lookup is $1/(1-\alpha)$, or a bit less for a successful lookup. Thus, if our load factor is 0.8, the expected number of probes per insertion will be $1/0.2 = 5$. This means that the table will work fairly fast for us with only a moderate amount of unused space.

Related to this: by comparison with the bucket list approach, hash-probe table are economical on space because there is no need for all the references—either from the hash table to the start of the linked lists, or within the linked lists themselves. All we have to store in memory are the actual keys or key-value pairs we're interested in. One way to look at that is that the memory this saves us can be 'spent' on increasing the size of the table, and therefore decreasing the load factor—which in turn will decrease the expected number of probes, and so make everything run faster. So for the same amount of memory, we can get faster lookup with a hash-probe table than with bucket lists.

There is, however, a significant downside to the hash-probe scheme: supporting a **delete** operation is not so straightforward. The naive approach to implementing **d***elete*(k) would be simply to use the probe sequence for k to locate the array index $j = \sharp(k, i)$ at which k appears (if it does at all), then set $A[j]$ to 'empty' (e.g. $A[j] = -1$ in our above example). However, there is a problem with this: suppose we try to look up a key k and explore the probe sequence as far as some empty cell $A[j]$. Then we can no longer safely conclude that k is not present in the table at all: after all, perhaps $A[j]$ was occupied at the time when k was inserted, so that k was actually placed in another cell further down the probe sequence, but the item in $A[j]$ was later deleted. We can of course continue our search further down the probe sequence for k in the hope of finding k—but how will we know when to stop?

One way to address this problem is to put a special marker, known as a *tombstone*, into a cell whenever we delete an item, signalling that the cell used to contain an item which has now been deleted. For instance, we might use -1 as a marker for empty cells that have always been empty, and -2 as a tombstone marker for cells that are currently empty but were once occupied. When performing **insert**, we can stop probing whenever we reach a cell with -1 or -2, and insert our new entry there. However, when performing **contains** or **lookup**, we can stop once we reach a cell with -1 (and return 'key not present'), but if we reach a cell with -2 we should continue to the next hash code in the probe sequence. This means that all the operations will work correctly, although there is a risk that over the course of time, our table will fill up with tombstones which will slow everything down. There are things one can do to mitigate this, but in any case, things become more complicated than they are with just **insert** and **contains**.

Overall, hash-probe tables are a very competitive approach. Broadly speaking, this is how both sets and dictionaries in Python are implemented, with some slight differences in the memory layout which I won't go into here.

Solution 3: Perfect hashing

We end by very briefly mentioning a third approach to avoiding collisions, more radical than the first two, but which it's possible to use in certain specialized situations. This is the approach known as *perfect hashing*. The situations in which this is applicable are those in which we are working with just a *fixed set of keys*: we may wish to modify the value v associated with an existing key k, but we will never wish to insert a completely new key into the table.

The idea is that if the set S of keys is fixed, it's possible to tailor our choice of hash function $\sharp$ to this specific set S, in such a way that no collisions occur between two keys $k, k' \in S$. Such a function is known as a *perfect hash function* for S. Even if it takes some work to find such a function, this is something we'll only have to do once for the given set of keys, so it may be worth the investment of some computational effort. Of course, we would like the resulting hash function $\sharp$ to be fast to compute, and also for the information on how to compute $\sharp$ not to occupy too much memory. Having found a perfect hash function, we can of course store all our set or dictionary items directly in a single array A, with the entry for key k living in the cell $A[\sharp(k)]$.

We won't go any further into the details of perfect hashing schemes here, but an example of such a scheme is explored in depth as part of Coursework 1.