Hash tables

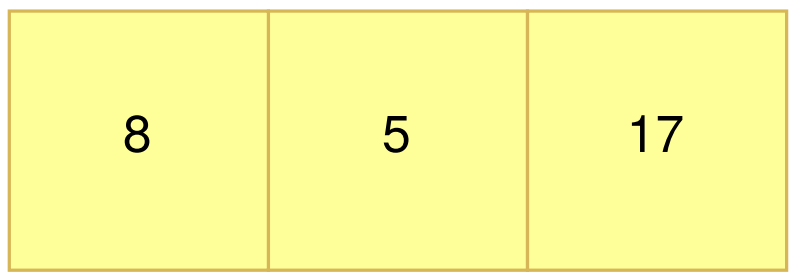
Lecturer: Arran Stewart

Outline

- What is a hash table?
- Implementing hash tables

Review

- Before we look at hash tables, let's first review the pros and cons of some simpler data structures.
- Recall that **arrays** allow us to store elements of a single data type contiguously in memory.
- We can access any element of an array in a single step by indexing into the array – array lookup has constant complexity, $O(1)$.
- This is very useful, because many algorithms depend on fast access to array elements
  - For instance, binary search, or sorting algorithms
  - It would be *possible* to do binary search on, say, a linked list, but very, very slow.

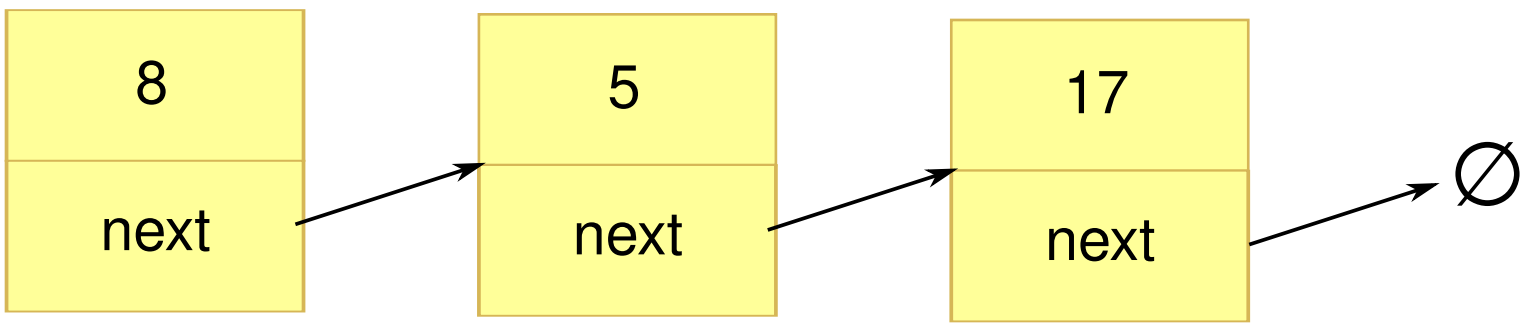


Review – arrays

- A downside of arrays is that their size is fixed – you must specify an array size when you create the array.
- There is no guarantee that more memory, adjacent to your array, will be available for use later.
- So arrays cannot easily grow.
  If you want an array of a new size, you have to create a fresh, new array of the desired size, and copy over all the elements of the old array (which has $O(n)$ complexity, where n is the size of the old array).
  - (Internally, this is what `java.util.ArrayList` does. If the array used by an `ArrayList` is full and more room is needed, a new array is allocated which in most implementations is 150% the size of the old one, then the contents of the old one are copied over. If you are interested, source code for one implementation is available here.)

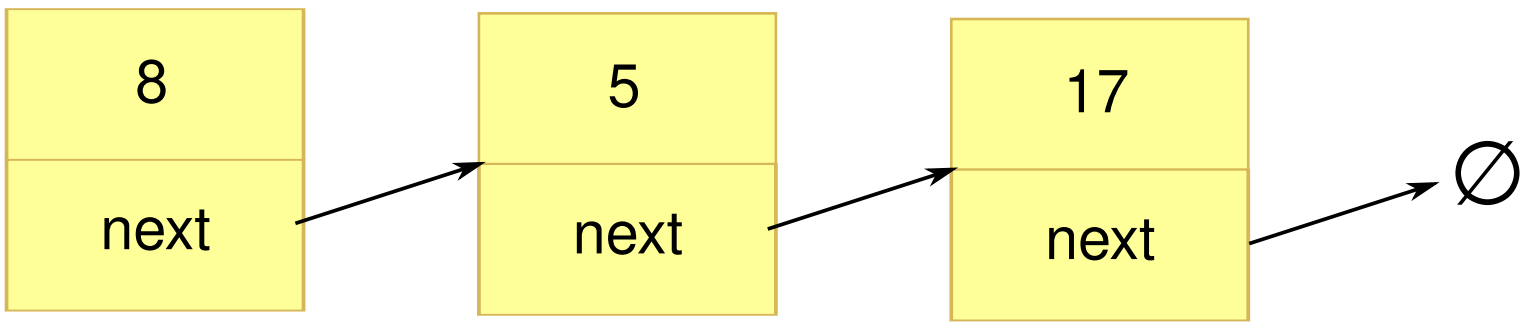
Review – linked lists

- Recall that we also learned about linked lists;
- Linked lists *can* can grow easily, because their elements are not necessarily contiguous in memory.
- Each node in a linked list contains the element that we want to store, and a pointer to the next node in the list.
  - when one node "points to" another node, they could be very far apart in memory.



Review – linked lists

- A disadvantage of linked lists is that we no longer have the ability to get quick access to individual elements via their index, like we did with arrays.
- If we want to access a particular element – say, the tenth item in a list – we need to traverse the first nine items in the list until we get to the tenth.
  So the complexity of lookup will be $O(n)$, where n is the index of the element we want to access.
- Or, if we want to find a list element containing the value "17", say – we must do a sequential search through the list, checking each node to see if it contains that value.
  - In the worst case, searching for an element has $O(n)$ – which is not very efficient.



Hash tables

- But what if we could combine the two in some way?
  What if we could have fast operations on a collection, like an array, but also allow our collection to grow?

Hash tables

- But what if we could combine the two in some way?
  What if we could have fast operations on a collection, like an array, but also allow our collection to grow?
- **Hash tables** offer a solution.

Hash tables

- But what if we could combine the two in some way?
  What if we could have fast operations on a collection, like an array, but also allow our collection to grow?
- **Hash tables** offer a solution.
- Hash tables are used when speedy insertion, deletion, and lookup of elements is a priority.

Hash tables

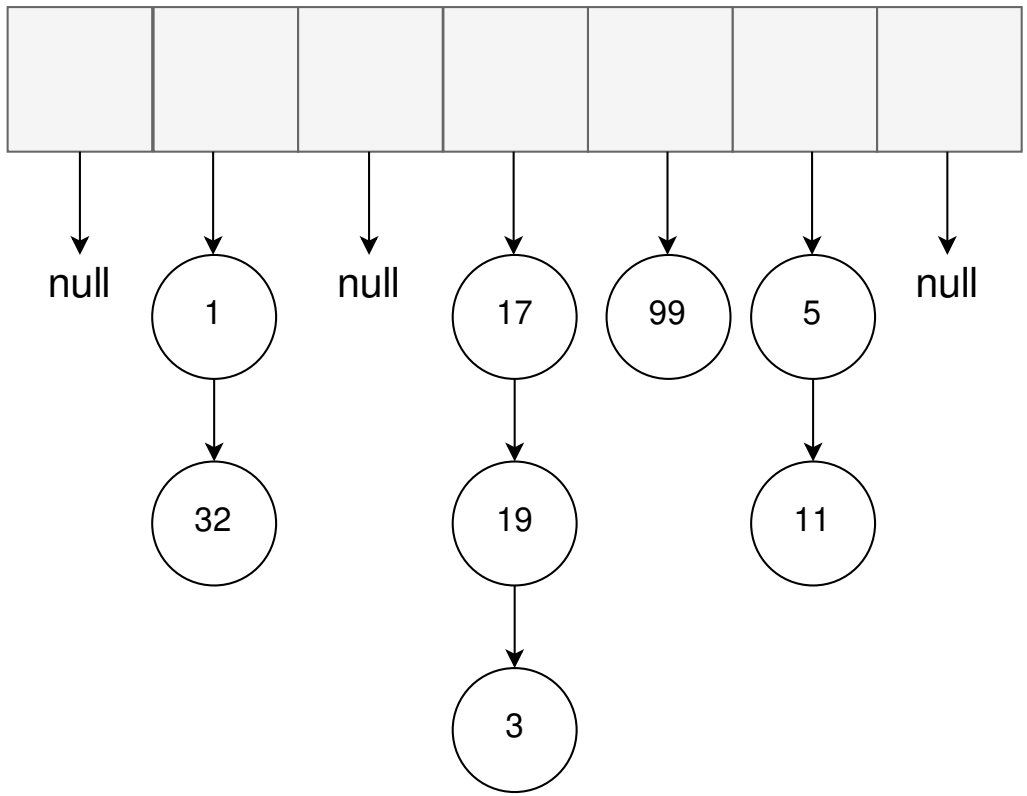
- But what if we could combine the two in some way? What if we could have fast operations on a collection, like an array, but also allow our collection to grow?
- **Hash tables** offer a solution.
- Hash tables are used when speedy insertion, deletion, and lookup of elements is a priority.
- In theory, insertion, deletion, and lookup can even be accomplished in constant time, $O(1)$.

Combining arrays and linked lists

- How can we combine arrays and linked lists?

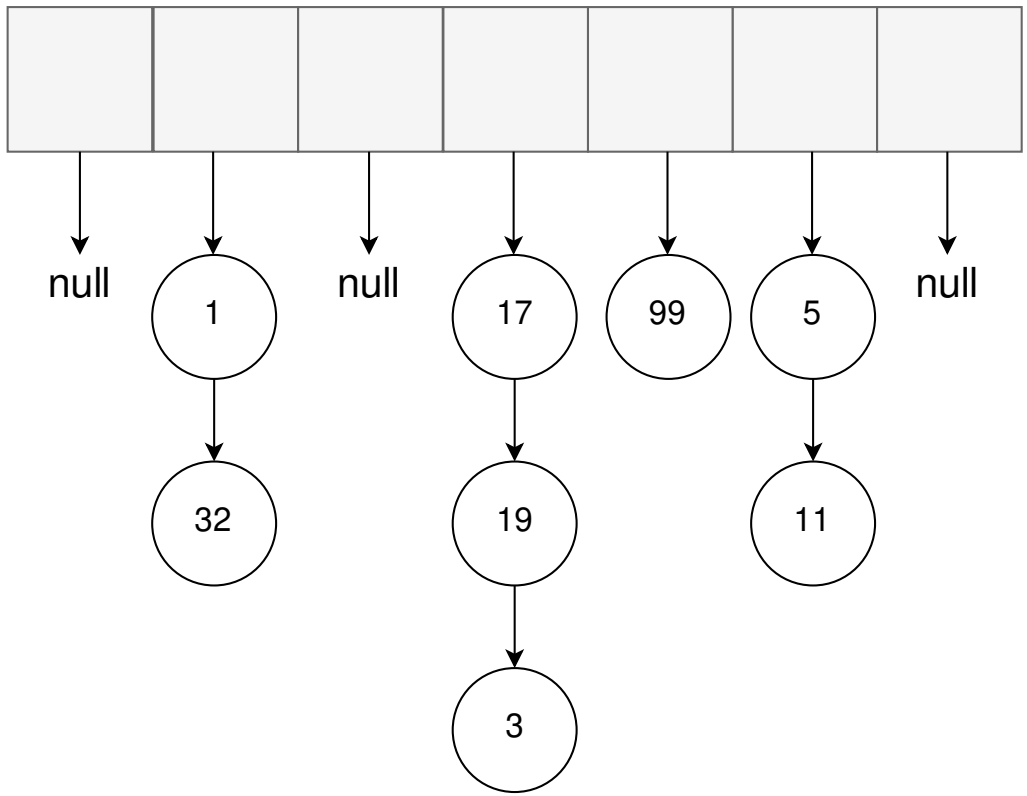
Combining arrays and linked lists

- How can we combine arrays and linked lists?
- What about creating an *array* of *linked lists*?



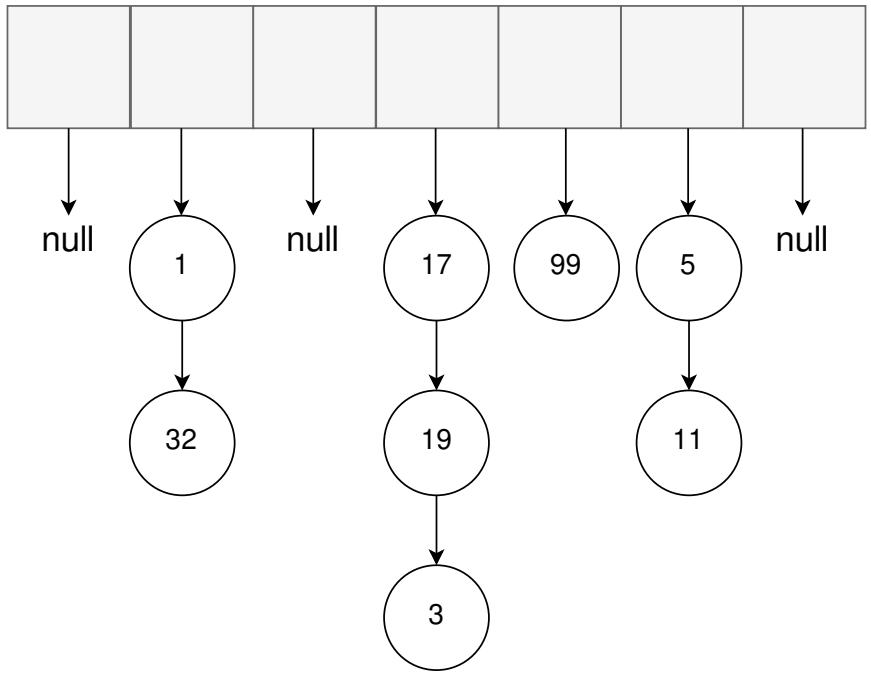
Combining arrays and linked lists

- How can we combine arrays and linked lists?
- What about creating an *array* of *linked lists*?



- As long as our lists were not too long, we would be able to get to any item we wanted quickly.

Combining arrays and linked lists



- For instance, if we ensured our lists were always at most 5 items long – then we could quickly go to a particular cell in the array, and then traverse at most 5 elements to get to the one we wanted.
- But how will we organize our values so we can find a particular one in this structure?

Hash functions

- To do this, we rely on *hash functions*.
- What is a hash function?
  - It is a function that takes a piece of data as input – we'll call this the *key* – and outputs an integer, commonly referred to as a hash *value*.
- When given the same piece of data, the hash function *always* returns the same number.

$$hash(x) = ...$$

Hash functions

- What if we had a hash function, and could ensure that the numbers it output were always within the bounds of an array?
- For instance, if our array is of size 15, say, then we would be looking for a hash function which always output numbers in the range 0 to 15.

Hash function examples

Here is an example of a (not very good) hash function f that operates on strings:

- f: given some string s, look at the *first* letter of the string, and return a number based on the letter's position in the English alphabet.

So, for instance, `f("apple")` would be 0, because the letter `'a'` is the zeroth[1] letter of the English alphabet.

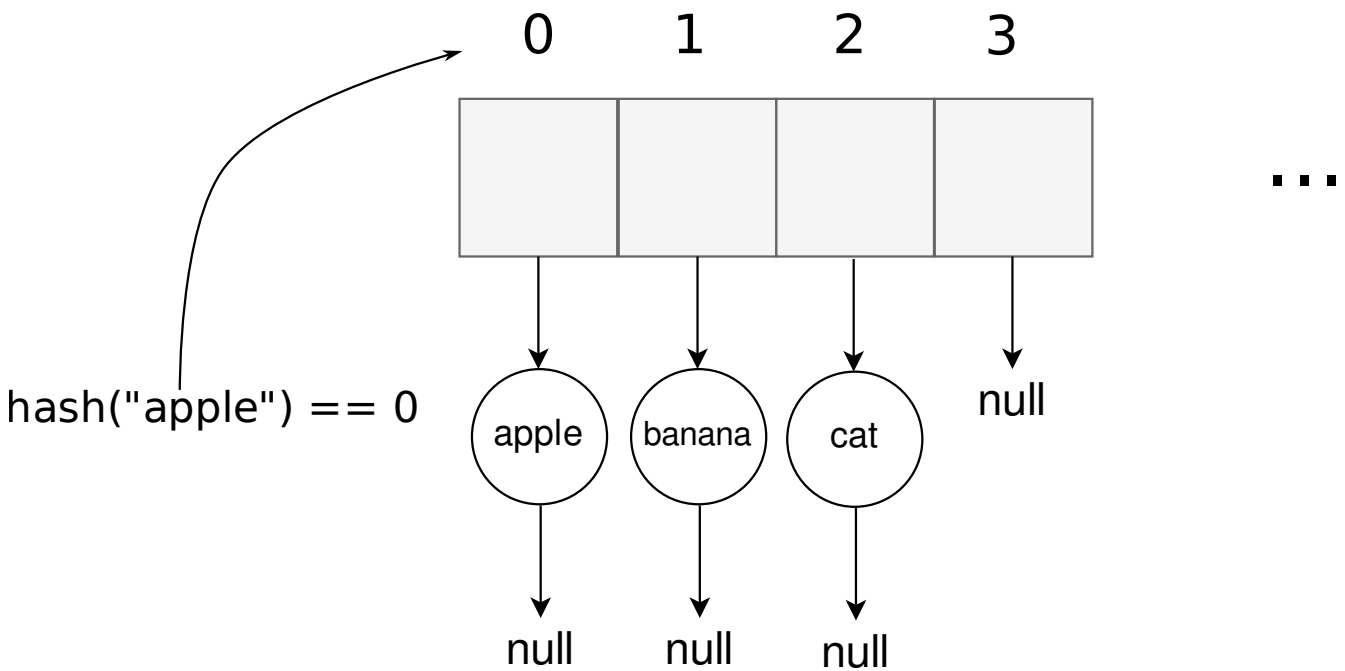
And `f("banana")` would return 1, and `f("cat")` would return 2.

[1] Many people might say it is the *first* letter. But in computer science, we nearly always prefer to start counting from zero.

Hash function examples

Now we could store these strings in an array:

- We would store `"apple"` in position 0
- `"banana"` in position 1
- `"cat"` in position 2
- and so on, if we had more strings.



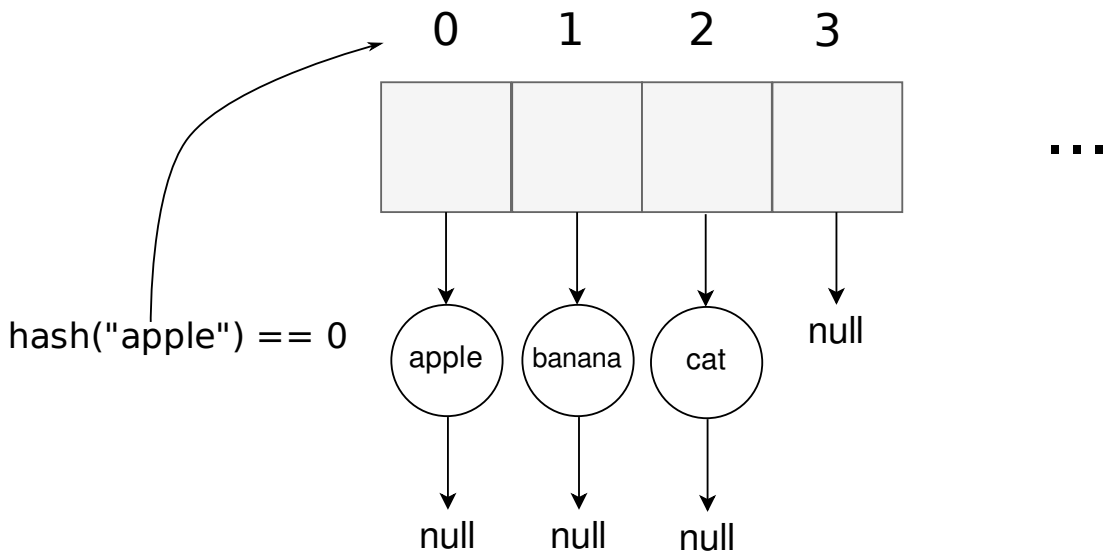
Hash function examples

This would also mean we could search for items very quickly.

If we wanted to know if the word “cat” is in the array, we just look at its first letter; see that it is “c”; and look in position 2 of the array.

If the array cell is empty (i.e. `null`), then the item is not in our collection.

So, for instance, we hash the string “dog”, the result is 3; but if we look in position 3, there is nothing there.



Hash function performance

- Now, looking at the first letter of a string is very fast.
- And accessing an array position is very fast.
- So this seems to mean we have fast, $O(1)$ access to items in our collection.

Hash function problems

A problem with this system: what if we have more than one word beginning with `'a'`?

Perhaps we want to store the word “ant” in our array; and `f("ant")` will give us 0, the same as for “apple”.

Separate chaining

- This is where the linked list comes in useful.

Separate chaining

- This is where the linked list comes in useful.
- Instead of storing each string *directly* in the array, we can make each cell of the array store a linked list.

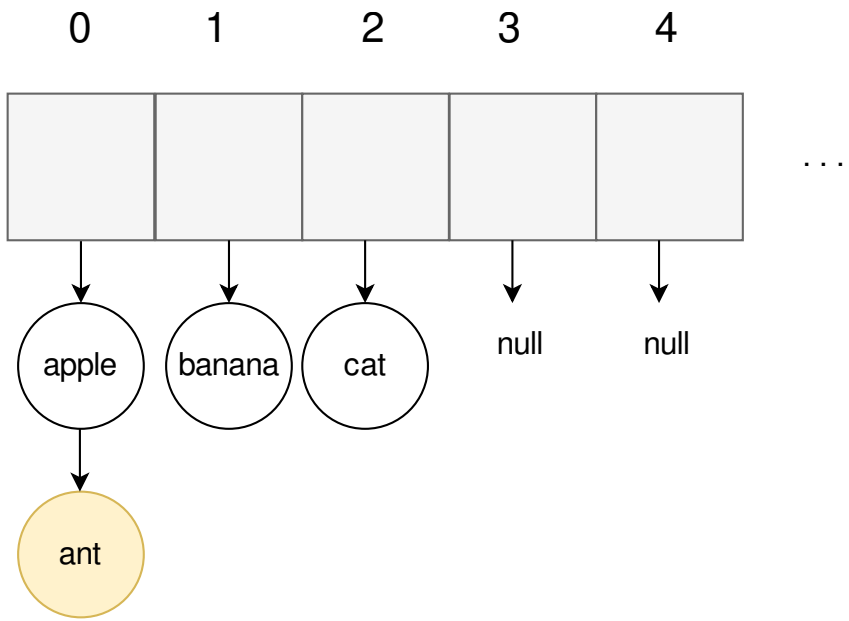
Separate chaining

- This is where the linked list comes in useful.
- Instead of storing each string *directly* in the array, we can make each cell of the array store a linked list.
- And if two items hash to the same position in the array - we simply *insert* a new item into our linked list.

Separate chaining

When we have two items which have the same hash value, that is called a *collision*; and using linked lists to resolve hash table collisions is an approach called *separate chaining*.

It looks like this:



Separate chaining performance

What is the performance for a hash table based on separate chaining?

Well – if we consider it based on the size of the longest linked list – let's call that number m – then the complexity of lookup is $O(m/k)$, where k is the size of our array.

Which, if we consider k to be constant, is the same as $O(m)$, which is linear, and not especially good.

Separate chaining performance

An example: suppose we use our simplistic "first letter" hash function, and that we have allocated an array of size 26 (one for each letter in the English alphabet).

If the longest chain happens to be of length 15, let that be m; then the complexity of lookup is $O(m/26)$.

And $O(m/26)$ is the same as $O(m)$. (We don't simplify any further, as we're expressing speed in terms of the size of the longest list.)

We've said in previous lectures that if we can get it, we'd much prefer $O(\log m)$ or $O(1)$, if we can get it.

Separate chaining performance

On the other hand, as long as our linked lists never grow too long, in practice, the performance could be very good.

If k is 100, say, and we have 104 items in our table, then at least a few cells must have more than one value stored in their linked list. But the results will be very *close* to the case where we have constant time access.

(An alternative to using linked lists is an approach called "linear probing": if we want to store "apple", and cell 0 is full, we just store the word in the next available cell. But this still results in linear search.)

Resizing

- In practice, hash table often adopt a method called "dynamic resizing" to deal with too many collisions.
- We could have a rule:
  No linked list in our table may ever be longer than 5 elements long.
  If any linked list is about to grow to more than 5 elements – we allocate a new array, with double the size of the old array, re-hash all the elements.
- This means we need a better hash function, though – our old hash function, f, only gave us numbers from 0 to 25, and now we need numbers from 0 to 51.

Improved hash functions

- In practice, we instead use hash functions with a very large range – say, from 0 to $2^{31} - 1$, the range of positive `ints` in Java – and just use the modulo (%) operator to work out which index of our array to use.
- So for instance, if we had a good hash function h for strings, which gave as `ints` in this range, we could use it as follows:
  - Given a string "apple", calculate `h("apple")`; suppose the result is 82347.
  - Calculate the modulus of 82347, given the size of our array (26) – the result of `82347 % 26` is 5.
  - So we store "apple" in position 5 of the array.
  - And if the array size doubles, we can re-calculate a new position for apple, based on the new size of the array.

Perfect hash functions

Some terminology:

A hash function that maps each item to a unique position is called **perfect**.

If we had an infinitely large array, we could always design a perfect hash function. But in practice, of course, this is not possible.

So instead, we settle for hash functions which are not necessarily *perfect*, but do have some desirable properties.

Properties of good hash functions

- A good hash function should make use of **all information provided** by a given key in order to maximize the number of possible hash values.
- This is why our function f was not a good hash function – it only examines the *first* letter of a string.
- For a *good* hash function, if we give it the strings "cat" and "caterpillar", it should give us different `int`s, even though both words start with "cat".

Properties of good hash functions

- A good hash function should **spread values evenly** across the hash table.
- This will reduce the length of linked lists should collisions occur.
- It's also a good sign if your hash value is capable of generating very **different hash values for similar keys**, making collisions much less likely.
  - For instance, if our hash function gives us very different results for "caterpillar" and "caterpillars" – that gives us some confidence it may be a good hash function.

Further reading

We will not look in detail at the algorithms used to calculate hash values in practice.

But the textbook (Weiss) gives some good guidance in this area.