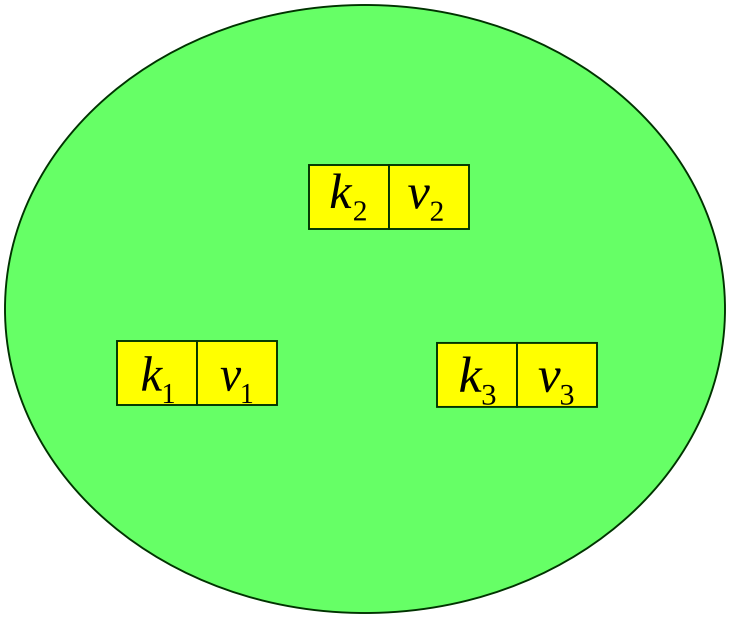
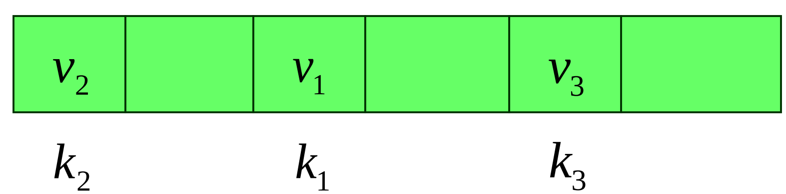
Map



If keys are integers in a small range, use an array indexed by key.



What if keys are from a large range or not even integers?

e.g. ISU ID (1000,000,000 for < 40,000 people)

Hash Tables

A *hash table* is a lookup table that acts as if it had random access into an array.

compute with a
hash function

1. Key ⇒ integer (*hash code*)

modulo
array size

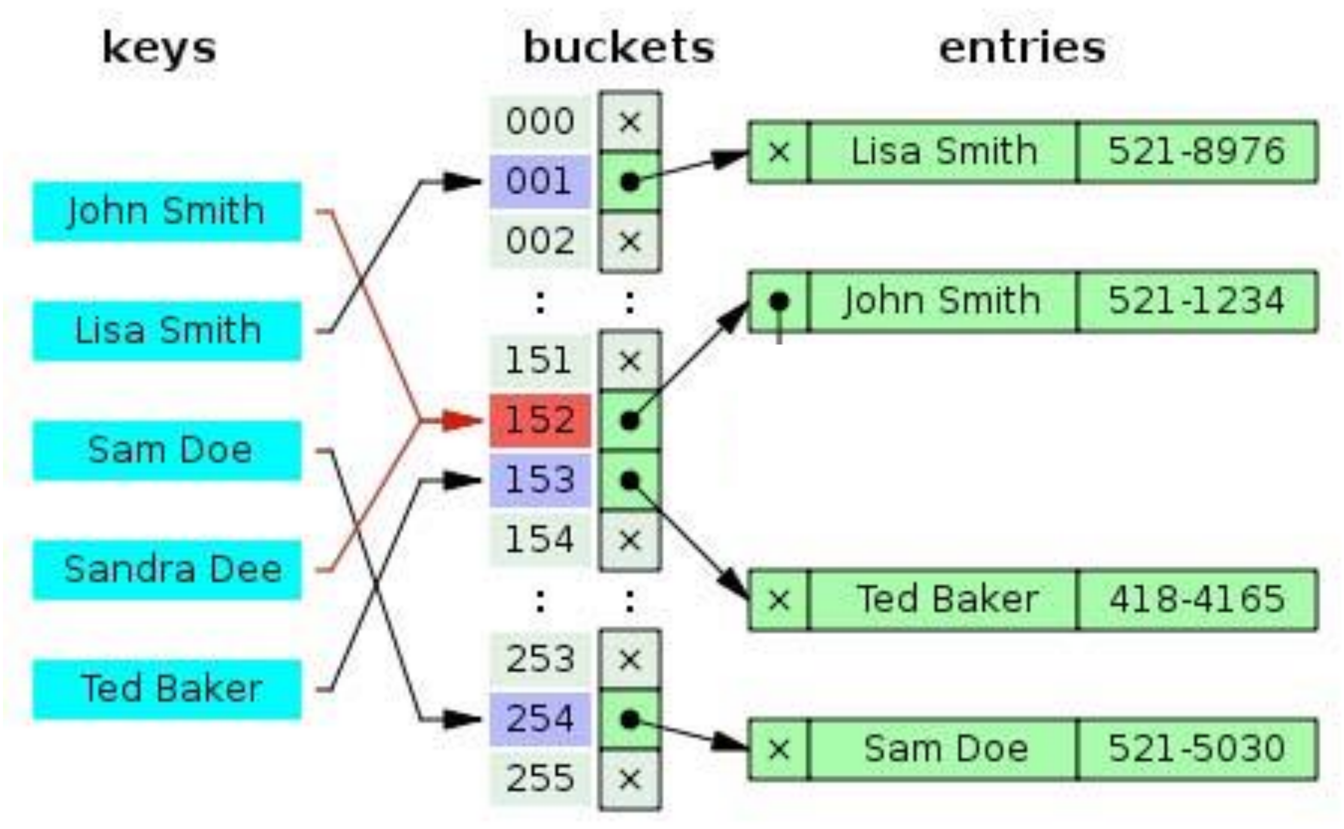
2. Hash code ⇒ index of an array

array size > # entries

3. Store the key and value in a linked list (*bucket*) of entries at the index. (This is called *chaining*.)

Buckets allow multiple values to be stored at the same index.

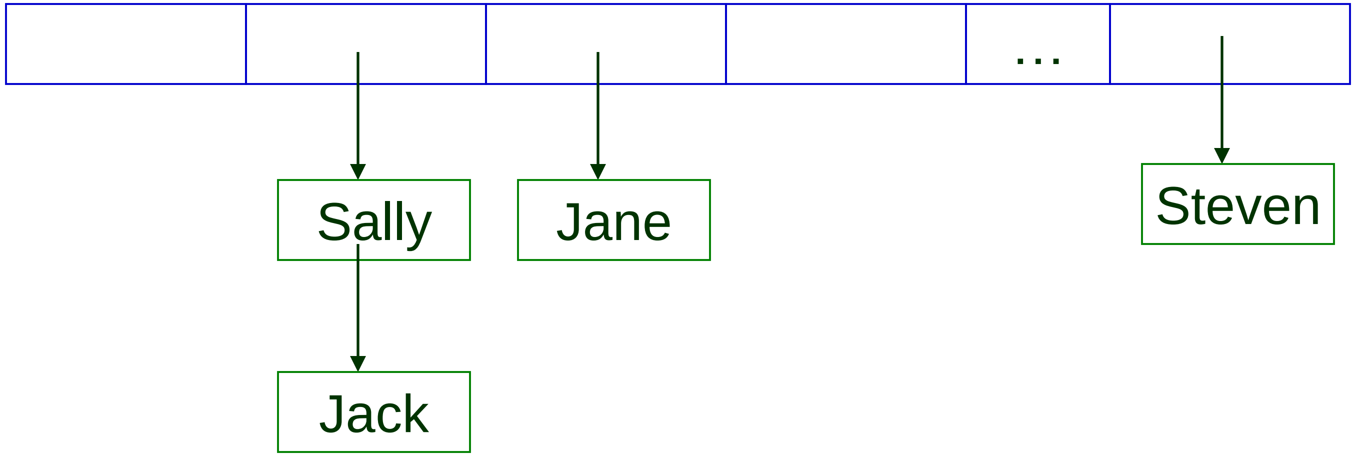
An Example from WikiPedia



Resolving Collisions

Collision happens when an item to be stored computes to the *same hash index* as an already existing item.

- Add to the linked list stored at the hash index.



Hash takes time $O(1)$ on the average if

- the data is well analyzed
- the *hash function* and table size are set to minimize collisions.

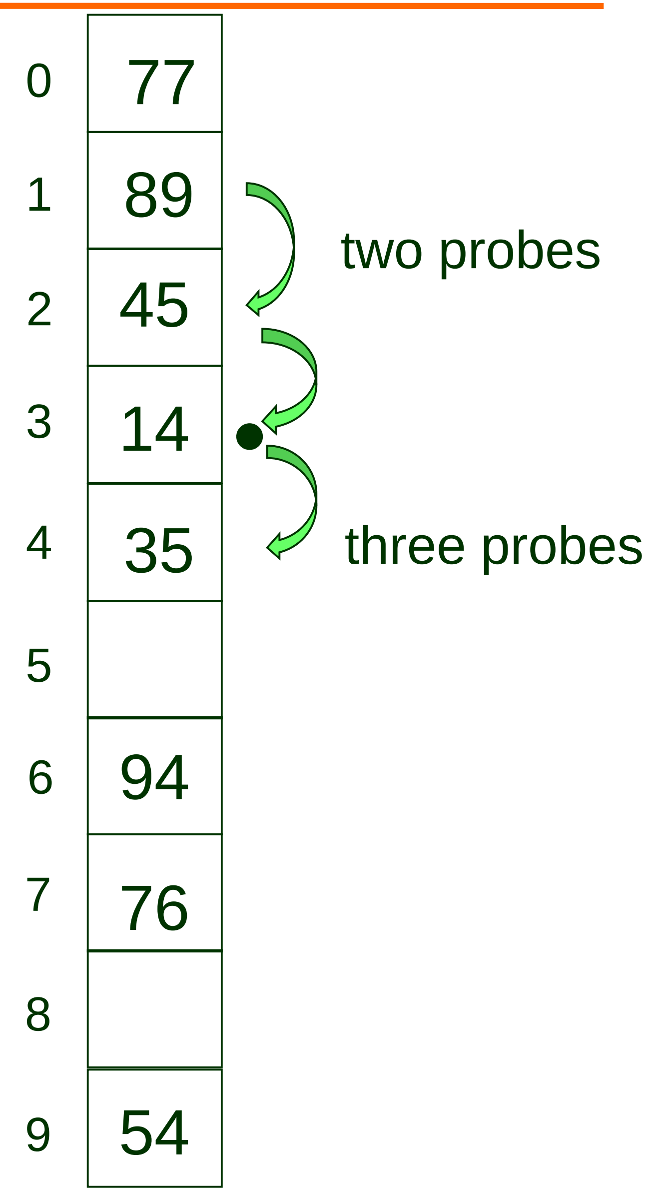
Linear Probing

Look for the first open slot.

- Compute the index using a hash function.
- If the location with the index is empty, then insert the item.
- Otherwise, look for the first open slot.
  - Starts at the next index.
  - Begins a sequential search at successive indices.
  - Insert the item at the first open location.

Example of Linear Probing

Hash Table



Key	Hash code
54	9
77	0
94	6
89	1
14	3
45	1
35	2
76	7

Hash Function Examples

In case we want to store

- ISU student records: a hash function may return
  - last few digits of ISU ID
  - last few digits of SSN
  - sum of char values on the student’s name
- A list of names
  - compute on the first 10 chars.
- Vehicle info for Iowa
  - convert letters in VIN to integers
  - convert letters in the license plate to integers

Hash Function in Java

Java implementations of hash tables: `HashMap` or `HashSet`.

Every Java Object has a default hash function `hashcode()`.

Example 1 Look up "John Smith" in a `HashMap` or `HashSet`:

- Calls the built-in `hashcode()`.
- Takes a modulus of the hash code to find the bucket.
- Searches the list for an entry whose key is equal to "John Smith", using `equals()`.

Example 2 HashMapDemo.java

Rules on Hash Code

1. Equal keys must have the same hash code.

2. If you override `equals()`, you must also override `hashcode()`.

It is highly desirable that if two keys are not equal, they have different hash codes.

Each bucket has length 1 (ideally) ⟹ $O(1)$ look up time.

Good Hash Function

- ♦ Deterministic – equal keys should produce the same value.
- ♦ Efficient to compute
- ♦ Uniformly distributing keys

Bad examples:

- ♠ Sum up ASCII values of the characters – high chance for collisions
- ♠ Use first three letters – words starting with some three-letter combinations are far frequent than others.

How to Write a Good Hash Function?

A good hash function will produce a value that incorporates *all the data* in the key.

Example

```
int result = some nonzero value
for each instance variable v used in equals()
    let c be an integer hashcode for v
    result = result * 31 + c
return result
```

prime number

Hash Code of a Variable

If v is

- an object, then c == *v*.`hashcode`().
- `short`, `char`, or `byte`: Set c = (`int`) v
- `boolean`: Set c = (v ? 0 : 1). i.e.; if v is true, c is 0, else it is 1.
- `float`: Set c = Float.floatToIntBits(v).
- `long`: Set c = (`int`)(v ^ (v >>> 32)).   (This does the XOR of the lower 32 bits with the upper 32 bits.)
- `float`: Set `long` x = Double.doubleToLongBits(v);
  c = (`int`)(x ^(x >>> 32)).

Hash Code of an Array

Apply the same rules to the individual elements in the array.

```
// hash code for String works like this
if (s.length() == 0)
   return 0;
int result = s.charAt(0);

for (int i = 1; i < s.length(); ++i)
{
    result = result * 31 + s.charAt(i);
}
return result;
```

Choice of Hash Function

- Should distribute keys uniformly into slots.
- Should not be affected by any patterns in the data.

Ex. Suppose keys are in the range [0,9999], and there are 100 slots.

Consider the hash function: $h(k) = k \,\%\, 100$

If you are given numbers that are all multiples of 100 to hash, they will all end up in the same slot 0!

Hash Function – Division Method

$$h(k) = k \,\%\, m$$

- Fast!
- Don't pick $m = 2^p$ or hash will depend on the *p* lower bits of k.
- Pick m as a *prime* not too close to a power of 2.

integer divisible only
by 1 and itself

Ex. 2000 character strings.

$$h(\text{“pt”}) = (\underbrace{112}_{\text{‘p’ in ASCII}} \bullet \underbrace{2^8}_{\text{8 bits per char}} + \underbrace{116}_{\text{‘t’ in ASCII}}) \;\%\; 1277 = 694$$

Multiplication Method

Choose a constant A with $0 < A < 1$ but close to 0 or 1.

Choose m as some power of 2.

For a key k, hashing in three steps:

1. β = the fractional part of kA.
2. $\beta = m\beta$ (left shifting)
3. $h(k)$ = greatest integer less than or equal to β (truncation)

Ex. $m = 8$ and 7-bit words.

$$
\begin{array}{lrl}
\text{Binary:} & .1011001 & A \\
 & 1101011 & k \\
\hline
 & 1001010.\underbrace{011}_{h(k)}0011 & kA
\end{array}
$$

1. Take the fractional part,
2. Discard the rest.
3. Shift it to the left.
4. Take the shifted out bits.

Load Factor

m buckets n entries

Rehash the table when it gets too full, more specifically, when

$$\frac{n}{m} > \alpha \quad \longleftarrow \quad \textit{load factor}$$

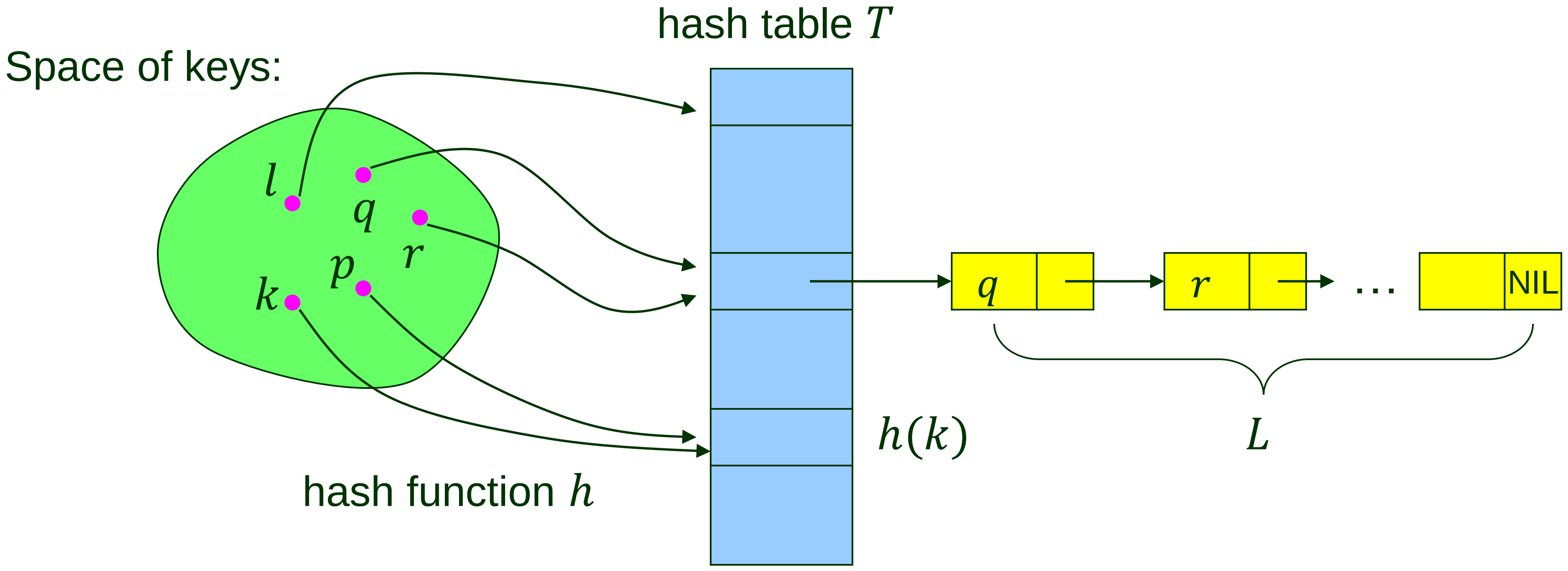
$\alpha = 0.75$ in Java

If α is exceeded, expand the table by increasing m until

$$m \approx 2n$$

A higher load factor decreases the space overhead, but increases the lookup time.

Operations on Hash Table



Insertion

Search

Deletion $\left\{ \begin{array}{l} \text{if singly linked list} \\ \text{if doubly linked list} \end{array} \right.$

Worst Case

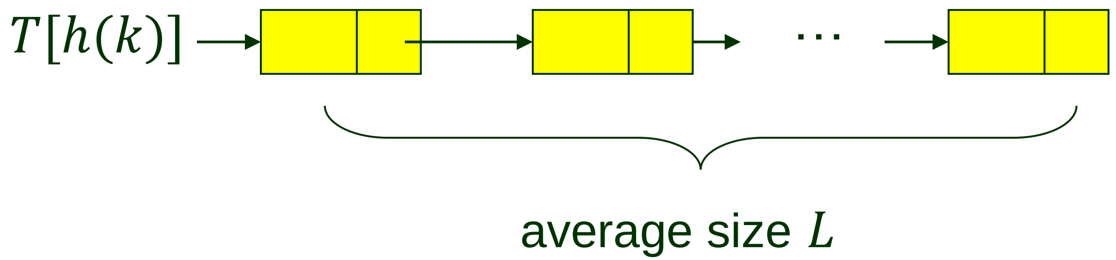
m slots n keys

Worst case: all keys hash to the same slot.

Search requires $O(n)$ time.

Simple Uniform Hashing

Every key k is equally likely to be hashed to any slot.



Time cost of unsuccessful search: $O(1 + L)$

- $O(1)$ for computing $h(k)$ and accessing $T[h(k)]$.
- Average time L to search to the end of one of the lists.

Time cost of successful search? $O(1 + L/2) = O(1 + L)$

- $O(1)$ to access the list.
- $O(L/2)$ to search the list.

Summary on Hash Tables

- Better than sequential search even when the data is *not* sorted.
- Require analysis of data characteristics beforehand.
- Require (possibly) more memory space.

A *hash table* stores and retrieves data item using a

$$\textit{hash function}\text{: data item} \rightarrow \underbrace{\text{a positive integer}}_{\text{address for storage}}$$

A perfect hash function produces a unique integer for every item.

But it may be *too slow* to compute.

So we often settle for less than perfect hash functions.