COMP20010

Data Structures and Algorithms I
11 - Maps and Hashing

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Maps

A map models a searchable collection of key-value entries

The main operations of a map are for searching, inserting, and deleting items

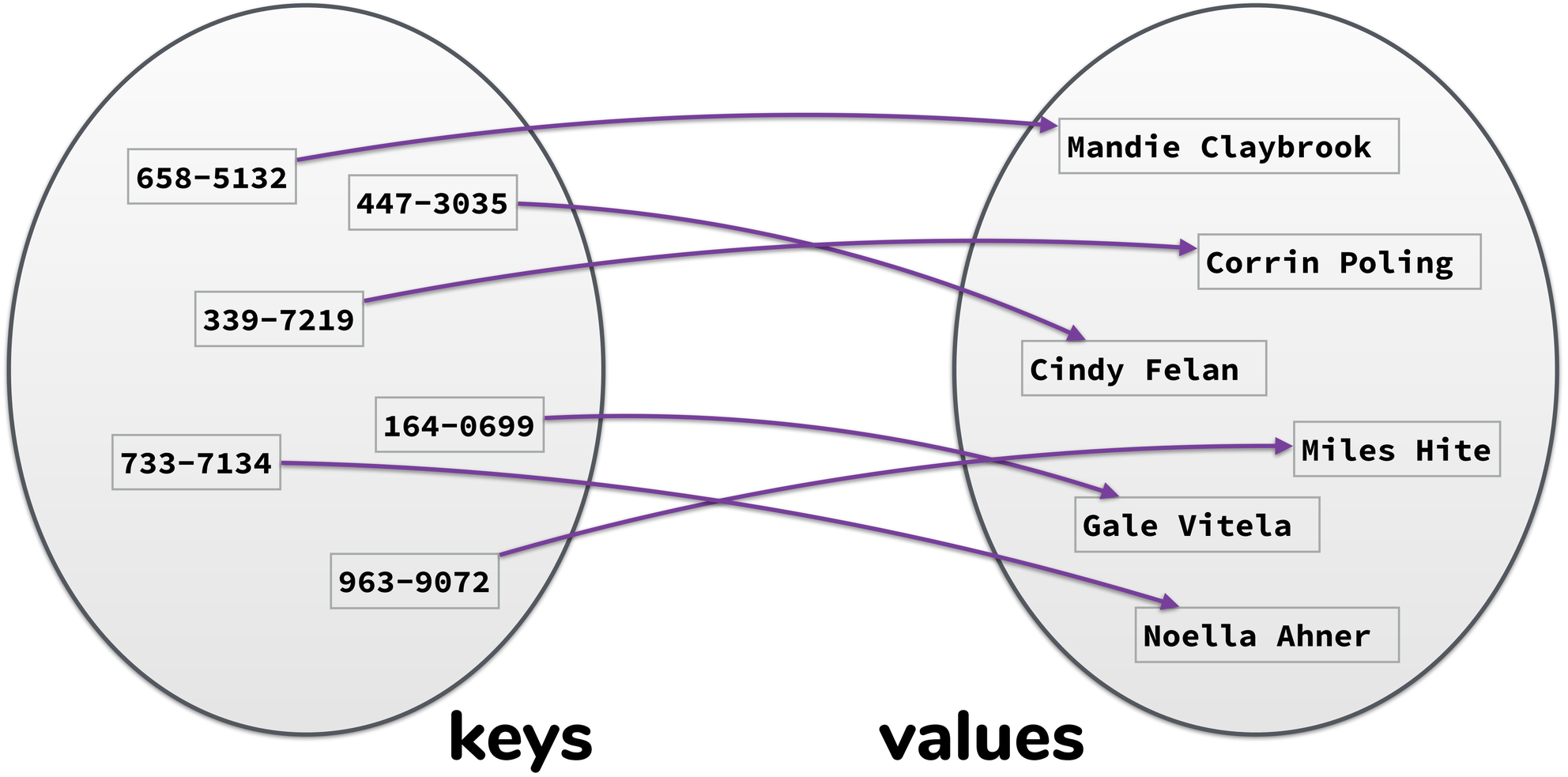
Multiple entries with the same key are not allowed

Applications:

- address book
- student-record database

»

Map


658-5132
447-3035
339-7219
164-0699
733-7134
963-9072
Mandie Claybrook
Corrin Poling
Cindy Felan
Miles Hite
Gale Vitela
Noella Ahner
keys
values

Maps

- The key is a unique identifier assigned to an associated object
- the key serves as an address for that value
- eg. student id’s -> we want them to reference the student records, also require them to be unique
- Maps are often referred to as associative data structures -> the key associated with the value determines the location in the data structure

Examples

```
#python

unwanted_chars = ".,-_ (and so on)"
wordfreq = {}
for raw_word in words:
    word =
raw_word.strip(unwanted_chars)
    if word not in wordfreq:
        wordfreq[word] = 0
    wordfreq[word] += 1
```

using python dict for counting words

```
<!-- Javascript -->

<!DOCTYPE html>
<html>
<body>
<p id="user"></p>
<script>
var s = '{"first_name" : "Sammy",
"last_name" : "Shark", "location" :
"Ocean"}';

var obj = JSON.parse(s);

document.getElementById("user").innerHTML =
"Name: " + obj.first_name + " " +
obj.last_name + "<br>" +
"Location: " + obj.location;
</script>
</body>
</html>
```

json object lookup

Map ADT

get(k)	if the map M has an entry with key k, return its associated value; else, return null
put(k, v)	insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
remove(k)	if the map M has an entry with key k, remove it from M and return its associated value; else, return null
keySet()	return an iterable collection of the keys in M
entrySet()	return an iterable collection of the entries in M
values()	return an iterator of the values in M
size()	number of elements in the stack
empty()	true if the stack is empty, false otherwise

Map Operations

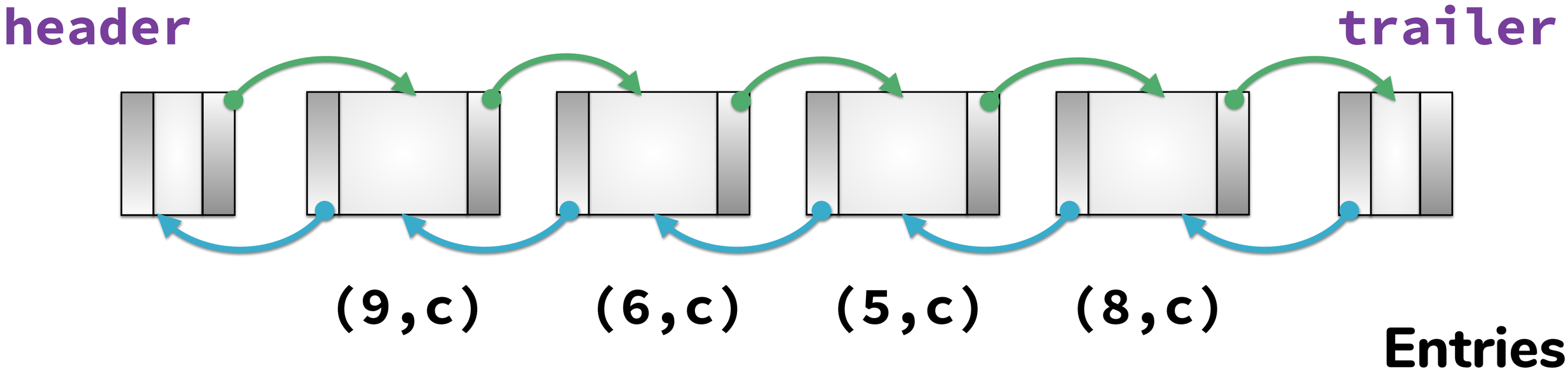
Operation	Output	Map
isEmpty()	TRUE	Ø
put(5,A)	null	(5,A)
put(7,B)	null	(5,A),(7,B)
put(2,C)	null	(5,A),(7,B),(2,C)
put(8,D)	null	(5,A),(7,B),(2,C),(8,D)
put(2,E)	C	(5,A),(7,B),(2,E),(8,D)
get(7)	B	(5,A),(7,B),(2,E),(8,D)
get(4)	null	(5,A),(7,B),(2,E),(8,D)
get(2)	E	(5,A),(7,B),(2,E),(8,D)
size()	4	(5,A),(7,B),(2,E),(8,D)
remove(5)	A	(7,B),(2,E),(8,D)
remove(2)	E	(7,B),(8,D)
get(2)	null	(7,B),(8,D)
isEmpty()	FALSE	(7,B),(8,D)

Map - List Based

We can implement a map using an unsorted list

We store the items of the map in a list S (based on a Doubly Linked list), in arbitrary order

The basic methods involves scanning through the list to find entry with key k.



Map - List Implementation

```
1:  function GET(k)
2:      Input retrieve the entry with key k from map
3:      B ← S.positions()                    ▷ B is an iterator of positions in S
4:      while B.hasNext() do
5:          p ← B.next()                          ▷ The next position in B
6:          if p.getElement().getKey() = k then
7:              return p.getElement().getValue()
8:          end if
9:      end while
10:     return ∅                          ▷ no entry with key=k, return null
11: end function
```

Map - List Implementation

```
1:  function GET(k)
2:      Input retrieve the entry with key k from map
3:      B ← S.positions()                    ▷ B is an iterator of positions in S
4:      while B.hasNext() do
5:          p ← B.next()                          ▷ The next position in B
6:          if p.getElement().getKey() = k then
7:              return p.getElement().getValue()
8:          end if
9:      end while
10:     return ∅                          ▷ no entry with key=k, return null
11: end function
```

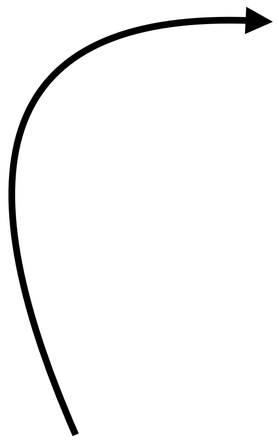
the essential point here is searching through the data structure for the key

Map - List Implementation

```
1:  function PUT(k, v)
2:      Input add the entry with key k and value v to map
3:      B ← S.positions()                    ▷ B is an iterator of positions in S
4:      while B.hasNext() do
5:          p ← B.next()                          ▷ The next position in B
6:          if p.getElement().getKey() = k then
7:              t ← p.getElement().getValue()
8:              S.set(p, (k,v))
9:              return t
10:         end if
11:     end while
12:     S.addLast((k,v))
13:     n ← n + 1          ▷ increment variable storing number of entries
14:     return ∅                        ▷ no entry with key=k, return null
15: end function
```

Map - List Implementation

```
1:  function PUT(k, v)
2:      Input add the entry with key k and value v to map
3:      B ← S.positions()                    ▷ B is an iterator of positions in S
4:      while B.hasNext() do
5:          p ← B.next()                     ▷ The next position in B
6:          if p.getElement().getKey() = k then
7:              t ← p.getElement().getValue()
8:              S.set(p, (k,v))
9:              return t
10:         end if
11:     end while
12:     S.addLast((k,v))
13:     n ← n + 1                  ▷ increment variable storing number of entries
14:     return ∅                          ▷ no entry with key=k, return null
15: end function
```



search for the element and then update its value

Map - List Implementation

```
1:  function REMOVE(k)
2:      Input remove the entry with key k from map
3:      B ← S.positions()                    ▷ B is an iterator of positions in S
4:      while B.hasNext() do
5:          p ← B.next()                          ▷ The next position in B
6:          if p.getElement().getKey() = k then
7:              t ← p.getElement().getValue()
8:              S.remove(p)
9:              n ← n - 1
10:             return t
11:         end if
12:     end while
13:     return ∅                          ▷ no entry with key=k, return null
14: end function
```

Performance of List based Map

Performance:

- put takes *O(n)* time since we can insert the new item at the beginning or at the end of the sequence- to avoid duplicates we need to check if the element is in the list
- get and remove take *O(n)* time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed

Arrays

```
int A[] = new int{0, 1, 2, 3, 4};
A[2] += 1;
```

we can access arrays by position

Arrays

```
public class Student {
	private String firstName;
	private String lastName;
	private String moduleCode;
	private String grade;
}
```

imagine a simple Student class

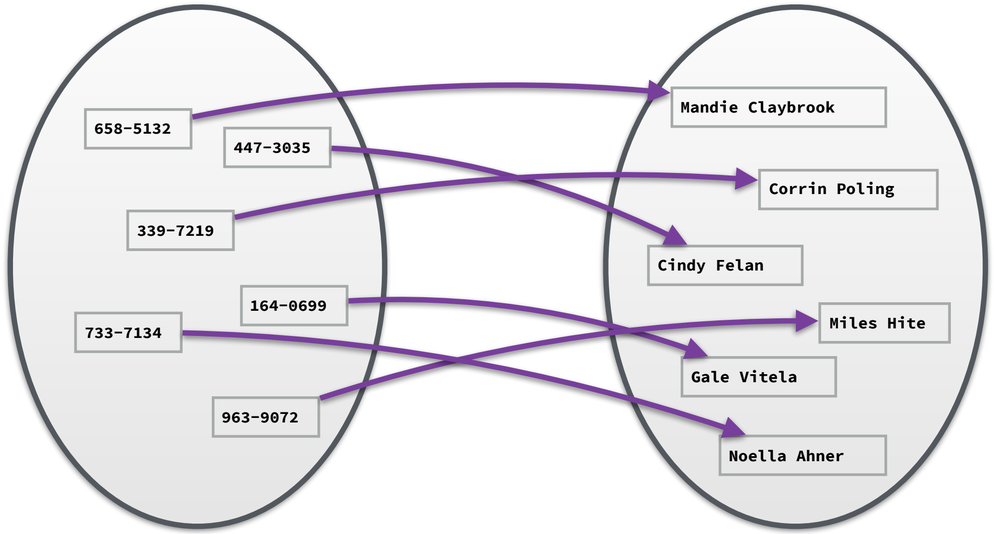
```
Student A[] = new Student{...};
A[“COMP20010”].setGrade(“B+”);
```

what data structure allows us to access the Student objects by a String?

Map

Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M.get(k) or M[k].

consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to N – 1, for some N ≥ n.



If the telephone numbers are integers, then we could make an array of size max(phone numbers), to guarantee a element for each number.

But this is very wasteful, we would only use a small portion in total.

But what should we do if our keys are not integers in the range from 0 to N – 1?

Hash Function

A hash function *h* maps keys of a given type to integers in a fixed interval [0, N - 1]

Example:

$h(x) = x \bmod N$

is a hash function for integer keys

h(x) is called the hash value of key *x*

A hash table for a given key type consists of

- Hash function *h*
- Array (called table) of size *N*

When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Hash Function

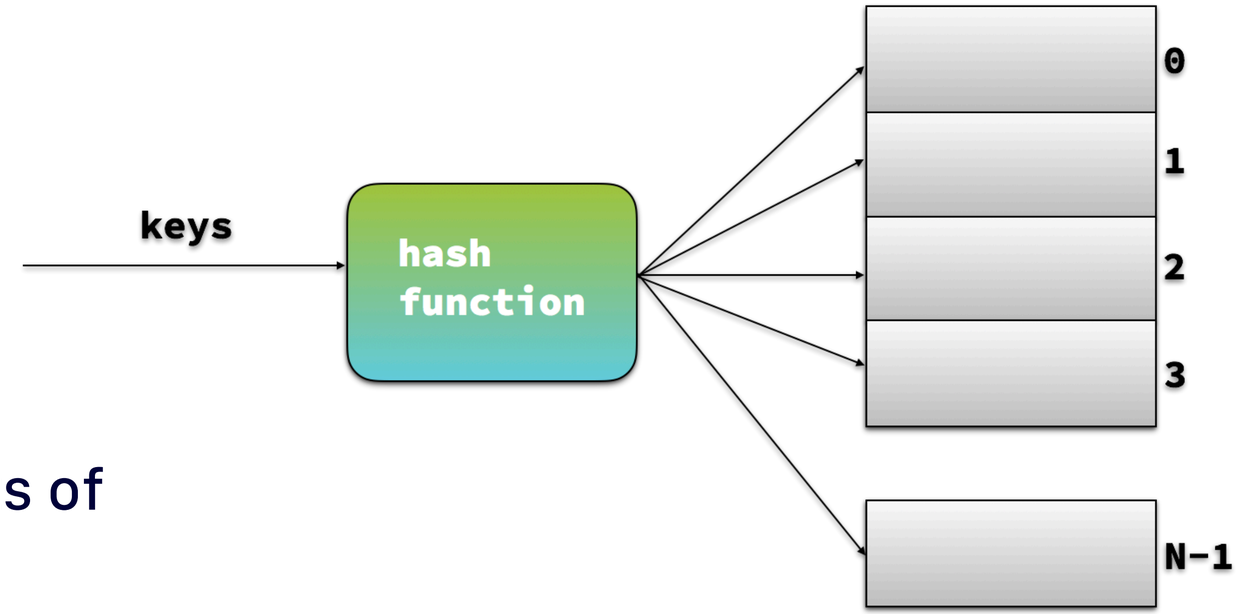
A hash function *h* maps keys of a given type to integers in a fixed interval [0, N - 1]

Example:

$h(x) = x \bmod N$

is a hash function for integer keys

h(x) is called the hash value of key *x*



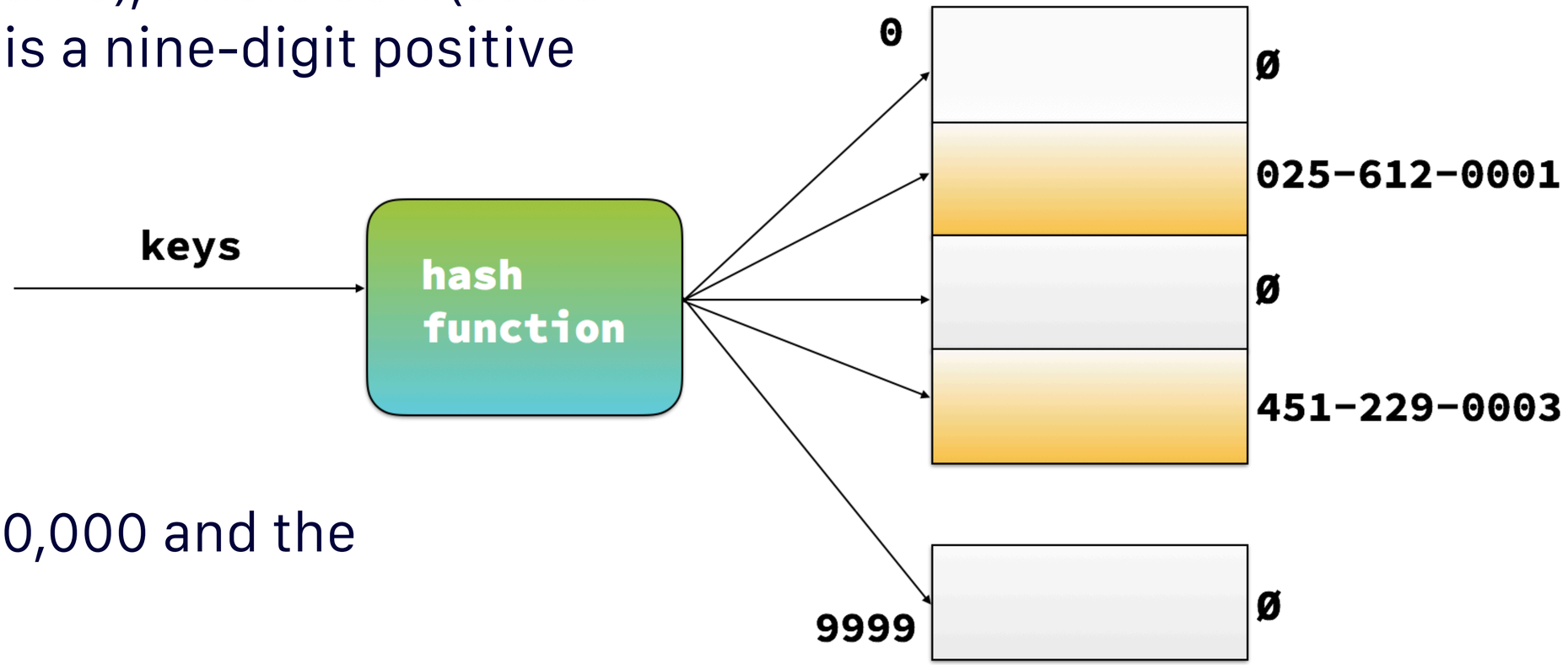
A hash table for a given key type consists of

- Hash function *h*
- Array (called table) of size *N*

When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Hash Function

We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer



Our hash table uses an array of size N = 10,000 and the hash function

$h(x)$ = last four digits of x

Hash Function

A hash function is usually specified as the composition of two functions:

Hash code:

h_1: keys → integers

Compression function:

h_2: integers → fixed range [0, N - 1]

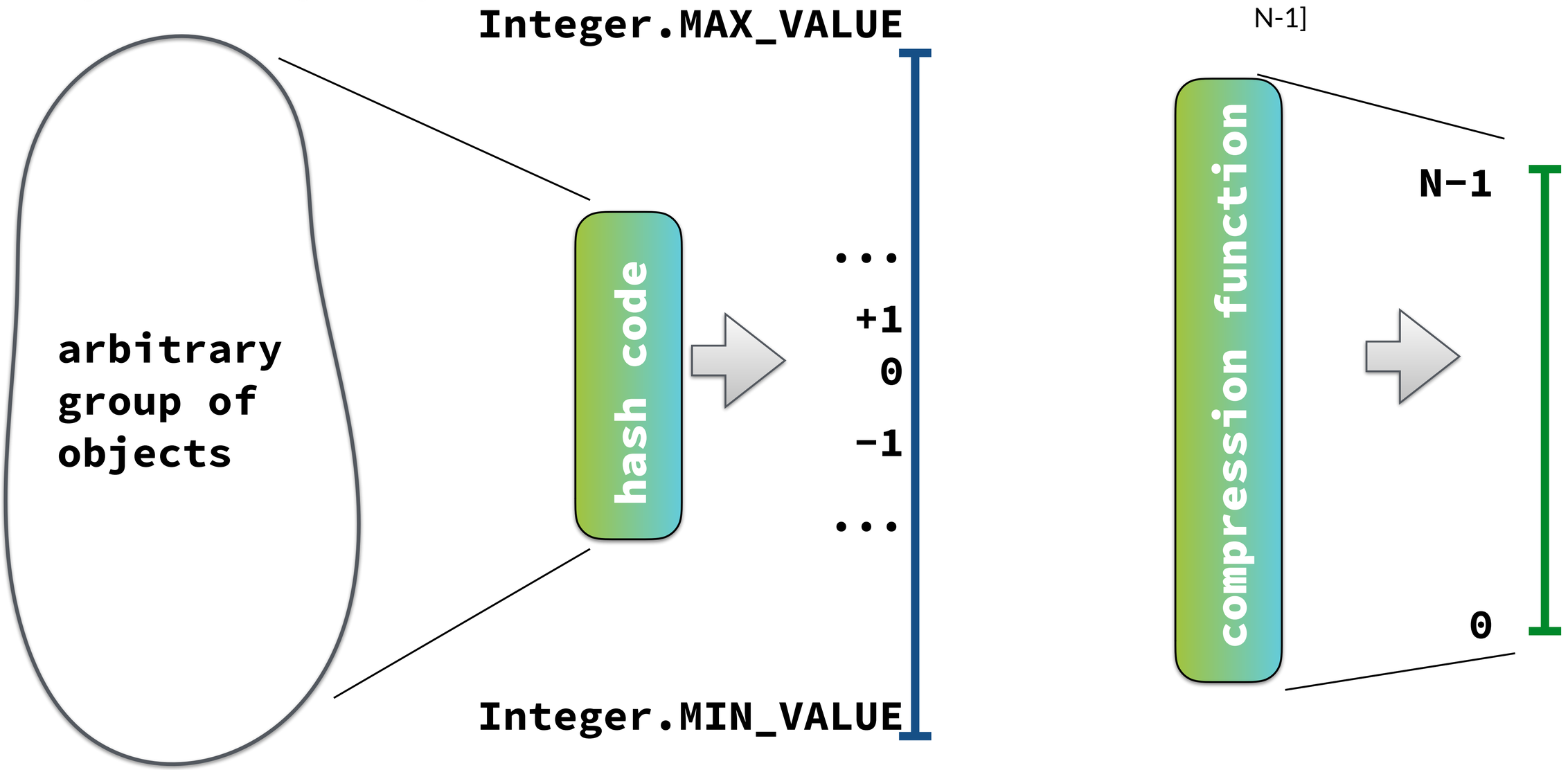
The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$

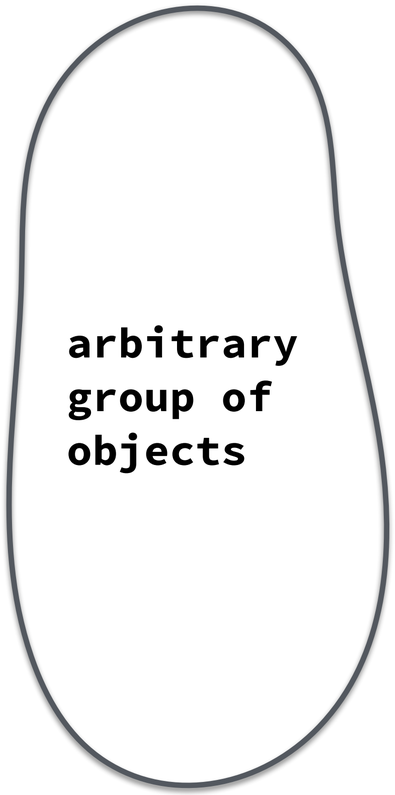
The goal of the hash function is to "disperse" the keys in an apparently random way

hash code maps keys of objects to integers (could be Integer or Long)

compression function maps the hash codes to the fixed range [0, N-1]



Hashtable



group by type

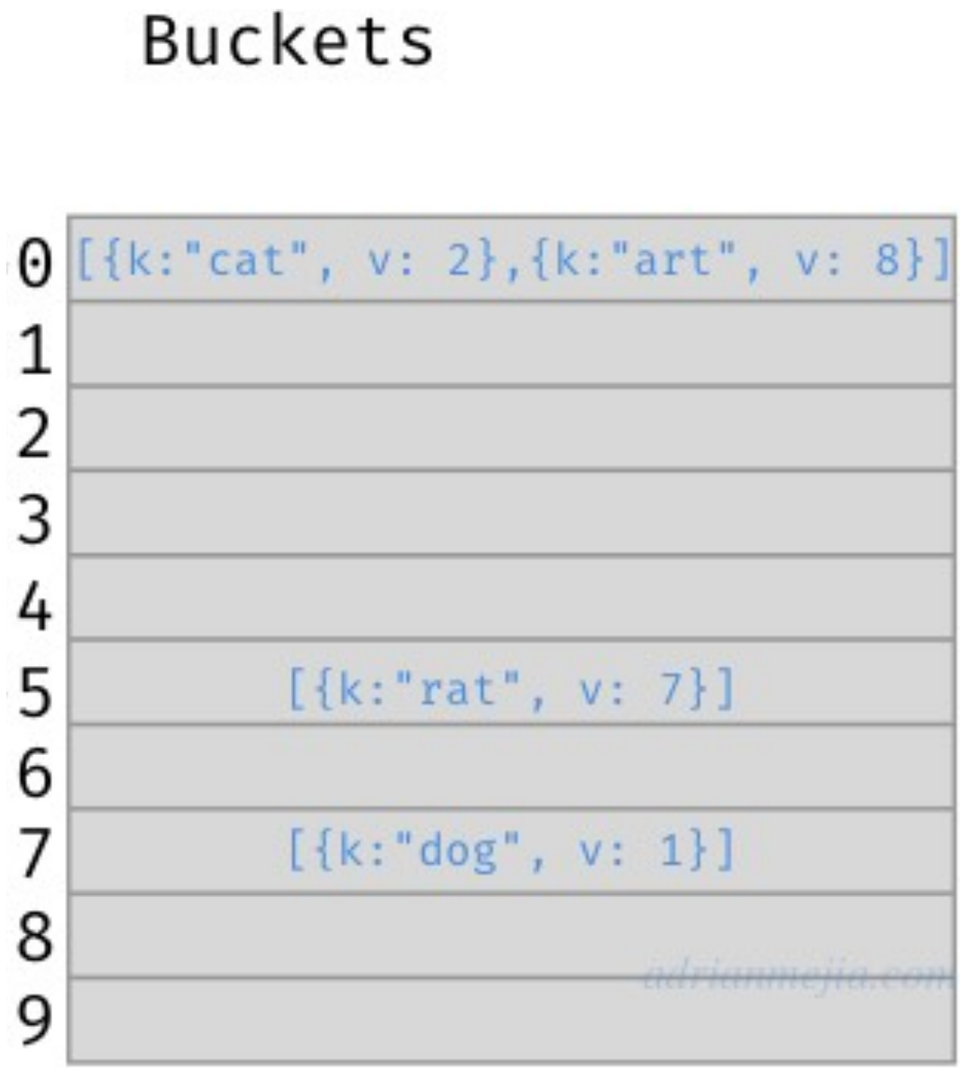
A Hashtable is a data structure that maps keys to values

searching in here is O(n)

but if you know the ‘label’ of the object, then you just look in the right drawer. Search is now O(1). A big improvement

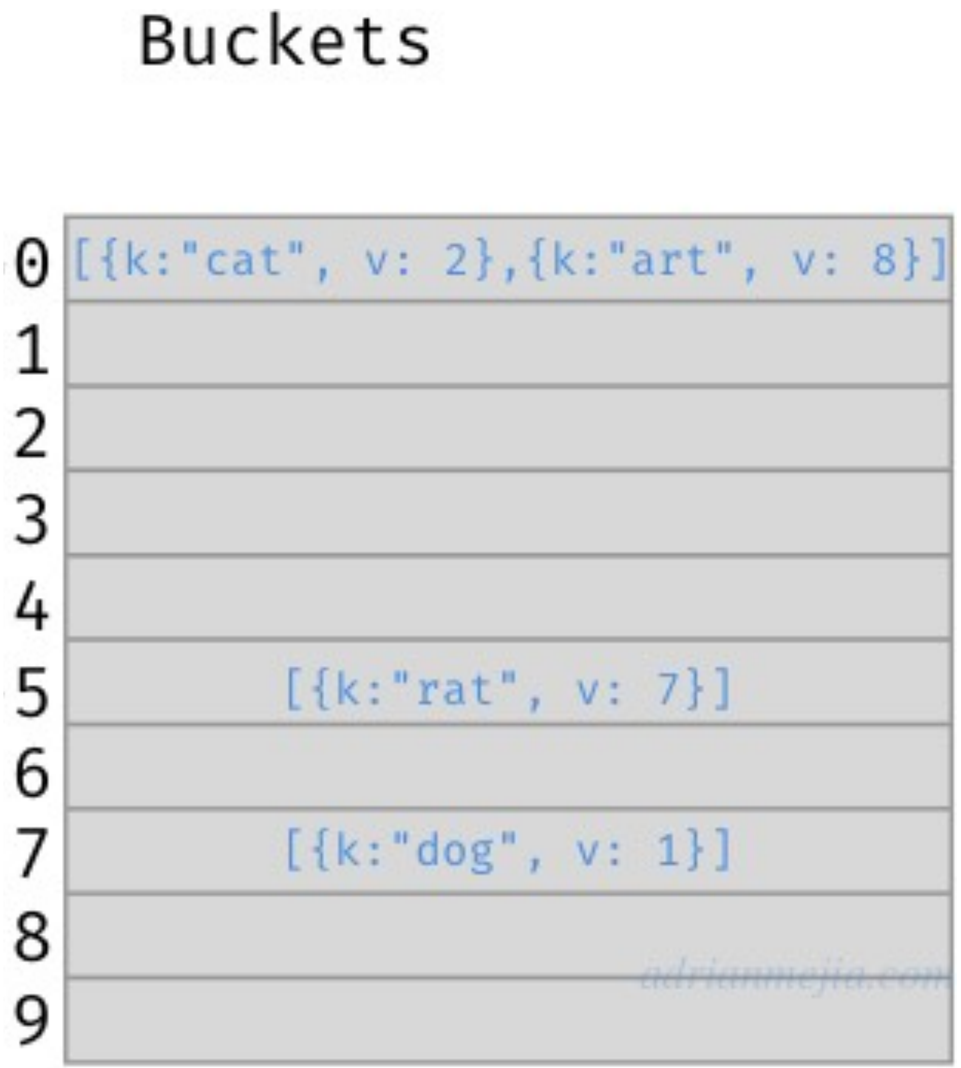
Bucket Array

- A bucket array is an array A of size N where each cell is a collection of (key, value) pairs
- If the keys are well distributed, all we need for a hash table is a bucket array
- An empty bucket can be a null object, otherwise we add an element with key k to the bucket at A[k]



Bucket Array

- if the keys are unique integers in the range [0, N-1], then each bucket will hold at most 1 element, and then searches, insertions, deletions will be O(1)
- But then the space required will be O(N), so if N is larger than the actual number of entries we need we are wasting a lot of space
- We need a good mapping from keys to integers in the range [0, N-1] and to choose the size of the bucket array carefully



Hash codes

Memory address:

- We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
- Good in general, except for numeric and string keys

Integer cast:

- We reinterpret the bits of the key as an integer
- Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)

Component sum:

- We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
- Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Polynomial Hash codes

Summing Components

- For any object x whose binary representation can be viewed as a k-tuple $(x_0, x_1, \ldots, x_{k-1})$ of integers:
- we can make a hash code for x as $sum_i(x_i)$

eg: a floating point number:

5437.12357

We could take as a tuple of
(5437, 12357)
and form the hash code from these two integers

In Java, the long or double type have double the bits of an integer. How do we map to integers?

- just cast to integer and ignore the high order bits
- sum the high order and low order bits

```
int hashCode(long i) {
    return (int)((i >> 32) + (int)i);
}
long i = 22337203685477580 7L;
(int) (i >> 32) = 52007855
(int) i = 494665727
hashCode(i) = 546673582
```

Polynomial Hash codes

- This summation approach does not work well when the order of the elements of the k-tuple ($x_0, x_1, \ldots, x_{k-1}$) is significant
- The problem is that when the objects share elements (like characters in a string), they map to the same hash code and we have collisions
- The alternative is use a scheme which accounts for the positions
- We will use something similar to the radix notation for an integer:

$$54321 = 5\times10^4 + 4\times10^3 + 3\times10^2 + 2\times10^1 + 1\times10^0$$

Polynomial Hash codes

- We want to partition the bits of the key into a sequence of components of fixed length (eg. 8, 16 or 32 bits)

  $x_0\ x_1\ x_2\ \ldots\ x_{n-1}$

- Then evaluate the polynomial

  $p(a) = x_0 + x_1\, a + x_2\, a^2 + \ldots + x_{n-1}\, a^{n-1}$

- for a fixed value of a and we ignore any overflows that occur

- This is a polynomial in a which takes the components of x ($x_0\ x_1\ x_2\ \ldots\ x_{n-1}$) as its coefficients

Polynomial Hash codes

- A naive way to evaluate a polynomial is to one by one evaluate all terms.
- First calculate x^n, multiply the value with a, repeat the same steps for other terms and return the sum. Time complexity of this approach is $O(n^2)$ if we use a simple loop for evaluation of x^n.
- We can use Horner's method to evaluate the polynomial more efficiently
- The idea is:
  - initialise the result as the coefficient of the highest power of x
  - repeatedly multiply result with x and add next coefficient to result.
  - Finally return result.

what is the Big-O complexity?

For the polynomial:
$2x^3 - 6x^2 + 2x - 1$

We can re-write it as:
$((2x - 6)x + 2)x - 1$

Horner’s Method

```
// Function that returns value of poly[0]x(n-1) +
// poly[1]x(n-2) + .. + poly[n-1]
static int horner(int poly[], int n, int x)  {
  // Initialize result
    int result = poly[0];

    // Evaluate value of polynomial using Horner's method
    for (int i=1; i<n; i++) {
      result = result*x + poly[i];
  }

  return result;
}
```

Cyclic Shift Hash Codes

- Instead of multiplying by a, we can use a cyclic shift of a partial sum by a fixed number of bits

```
static int hashCode(String s) {
  int h = 0;
  for(int i = 0; i < s.length(); ++i) {
    h = (h << 5) | (h >>> 27);
    h += (int) s.charAt(i);
  }
  return h;
}
```

5-bit cyclic shift of the remaining sum

Fine-Tuning

- The choice of parameters makes a big difference for our hash functions
- with polynomial hashing, a=33, 37, 39 or 41 work very well for Strings which represent English language words
- For any of these values the number of collisions on over 50,000 words is < 7.
- With a cyclic shift of 5 we find about 4 collisions on 25k words, and with a shift of 6 we get about 6 collisions.

Compression Functions

- The hash code for a key k returns an integer and is usually not suitable for use with our bucket array.
- We need to map the output of the hash code into the range [0, N-1]
- The mapping to the range [0, N-1] is the second important task for a hash function
- A good compression function minimises the number of possible collisions in a given set of hash codes.

Division Method

- We can simply divide the hash code by the range:

$$h_2(y) = y \bmod N$$

- The size of the hash table N is often chosen to be a prime number (why?)
- If the hash function is chosen well it should ensure the probability of two different keys getting hashed to the same bucket is 1/N.

eg: take keys from

[200, 205, 210, ..., 600]

and look at the h_2 values for each key

k % 100	
k	$h_2(k)$
200	0
205	5
210	10
215	15
220	20
225	25
230	30
...	

lots of collisions

k % 101	
k	$h_2(k)$
200	99
205	3
210	8
215	13
220	18
225	23
230	28
...	

no collisions

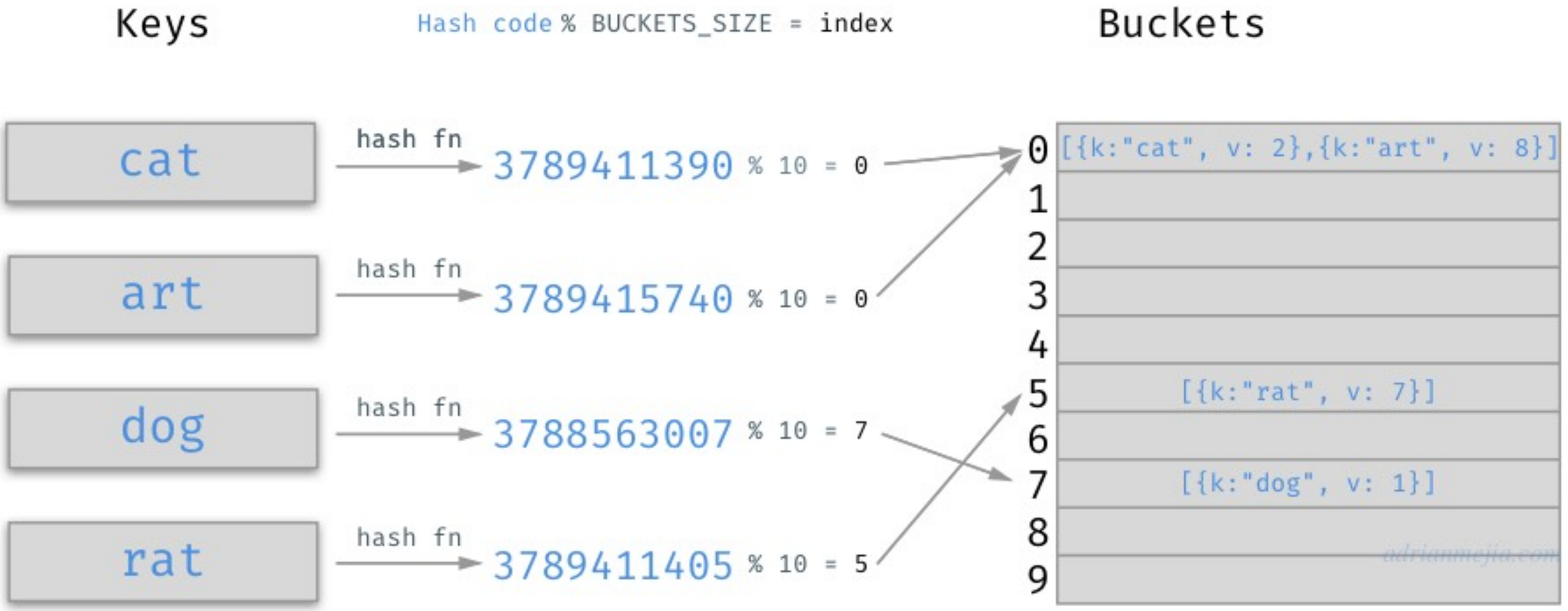
MAD (Multiply, Add and Divide)

- more sophisticated than the Division method
- avoids repeated patterns in the keys

$$h_2(y) = abs((ay + b) \bmod p) \bmod N$$

- where N is the size of our table, p is a prime number larger than N, and a and b are integers chosen at random from the range [0, p-1]
- Slightly more complex to implement, but gives a better distribution of hash codes

Hashcodes



Each key gets translated into a **hash code**. Since the array size is limited (e.g. 10), we have to loop through the available buckets using modulus function. In the buckets we store the key/value pair and if there's more than one we use a collection to hold them.

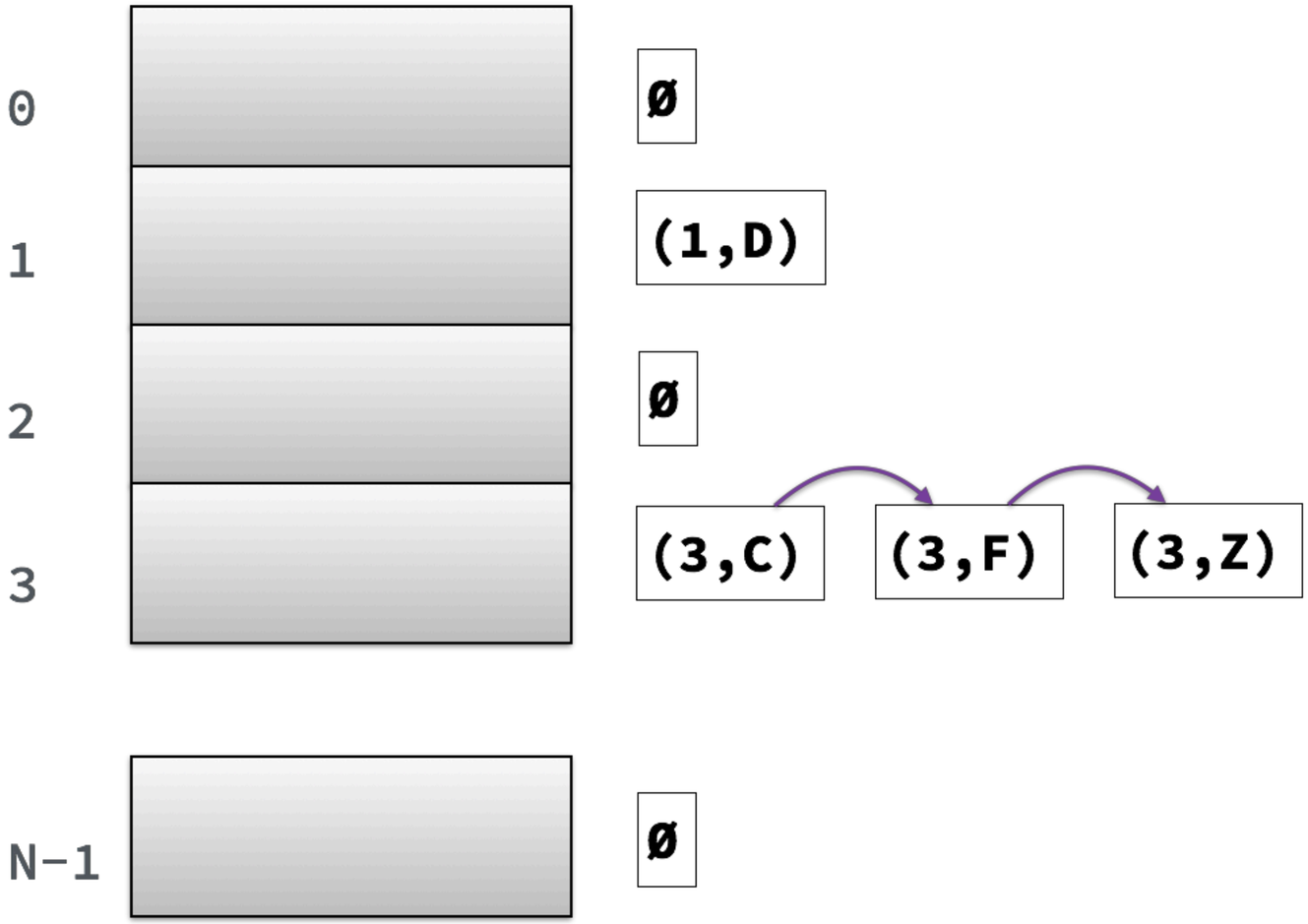
Collisions

Collisions

- Two keys mapping to the same location in the hash table is called “Collision”
- Collisions can be reduced with a selection of a good hash function
- But it is not possible to avoid collisions altogether
- There are methods to find perfect hash functions, but this is generally hard to arrange

Separate Chaining

- •**chaining:** All keys that map to the same hash value are kept in a linked list
- Separate chaining is simple, but requires additional memory outside the table



Separate Chaining

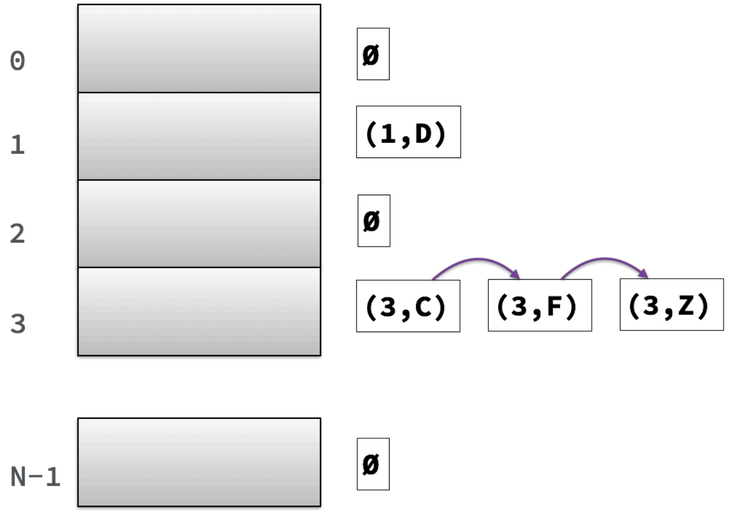
In the worst case, operations on an individual bucket take time proportional to the size of the bucket.

If we use a good hash function to index the n entries of our map in a bucket array of capacity N, the expected size of a bucket is n/N.

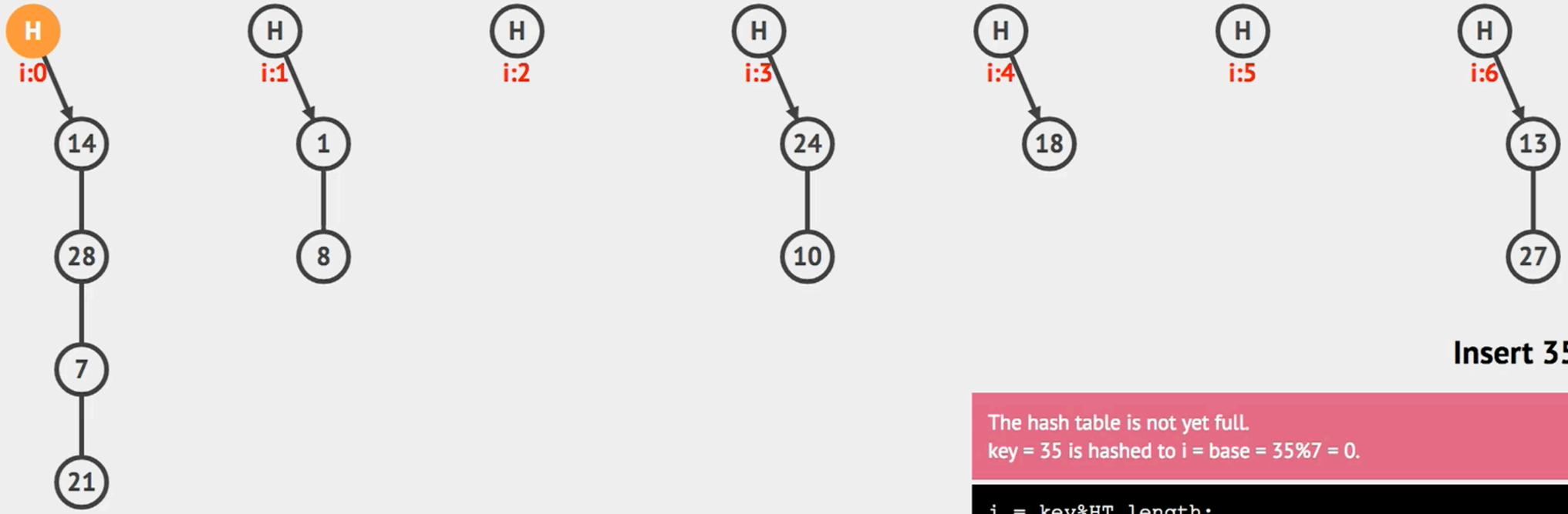
Therefore, if given a good hash function, the core map operations run in $O(\lceil n/N \rceil)$.

The ratio $\lambda = n/N$, called the *load factor* of the hash table, should be bounded by a small constant, preferably below 1.

As long as λ is $O(1)$, the core operations on the hash table run in $O(1)$ expected time.



VISUALGO.NET / en /hashtable LP QP DH **SEPARATE CHAINING** Exploration Mode Login

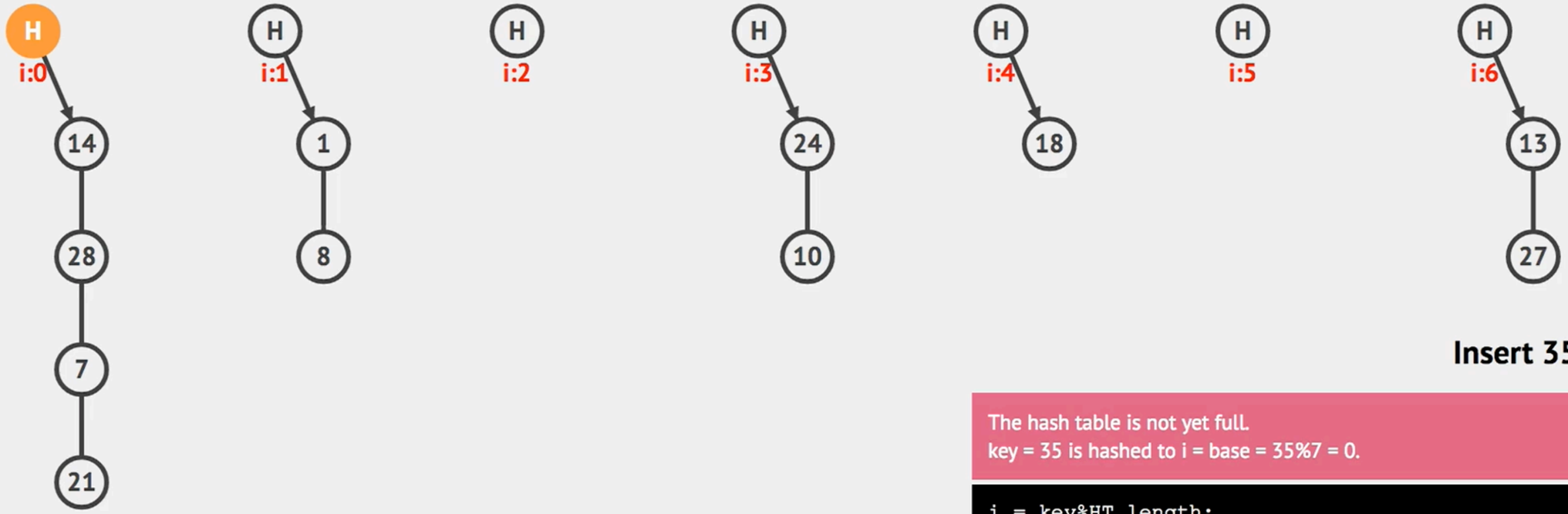


Insert 35,77

The hash table is not yet full.
key = 35 is hashed to i = base = 35%7 = 0.

```
i = key%HT.length;
if HT_SC[i].length == 6, prevent insertion
insert key to the back of this list i
```

VISUALGO.NET / en /hashtable LP QP DH **SEPARATE CHAINING** Exploration Mode Login



Insert 35,77

The hash table is not yet full.
key = 35 is hashed to i = base = 35%7 = 0.

```
i = key%HT.length;
if HT_SC[i].length == 6, prevent insertion
insert key to the back of this list i
```

Linear Probing

With separate chaining we have relatively simple implementations of hash map operations

slight disadvantage: It requires the use of an auxiliary data structure to hold entries with colliding keys

If space is at a premium then we can use the alternative approach of storing each entry directly in a table slot

This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to properly handle collisions.

There are several variants of this approach, collectively referred to as *open addressing* schemes

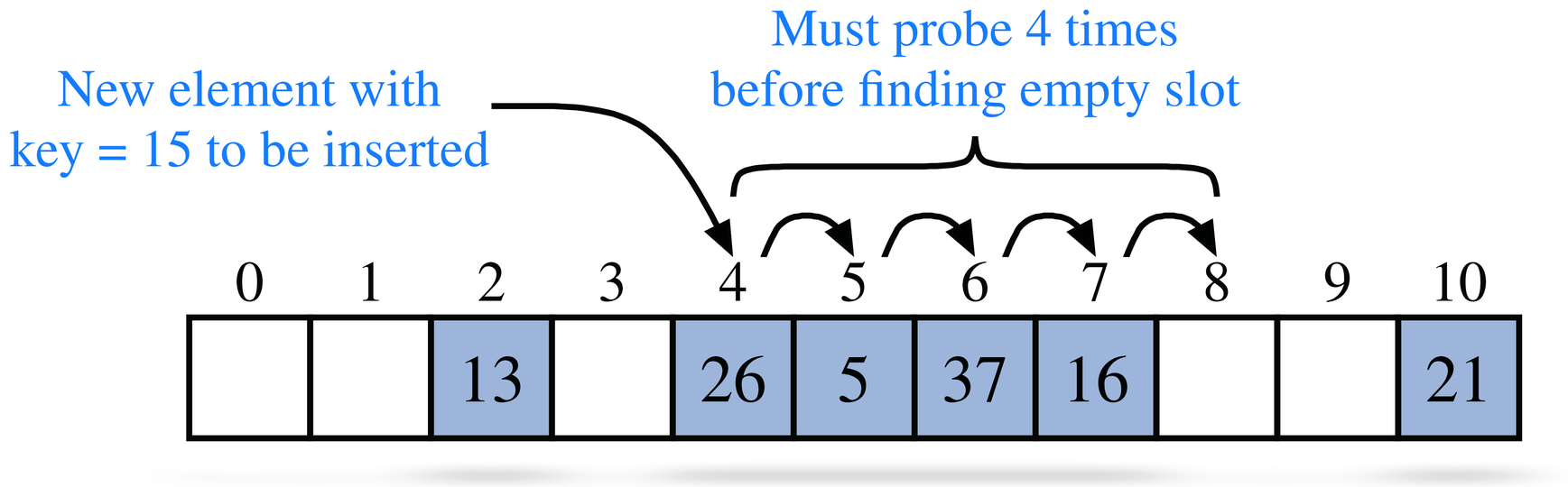
Open addressing requires that the load factor is always at most 1 and that entries are stored directly in the cells of the bucket array itself.

Linear Probing

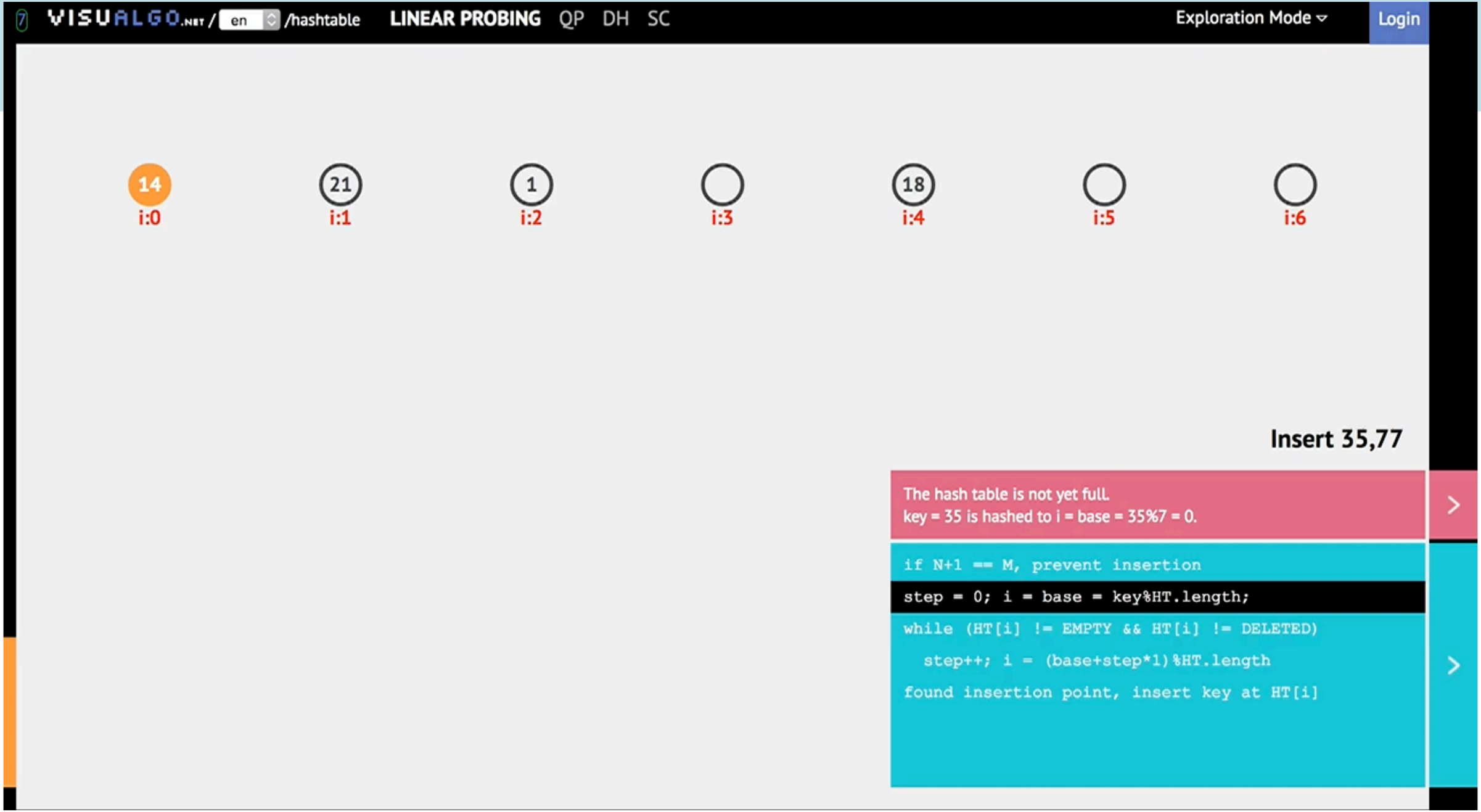
Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell

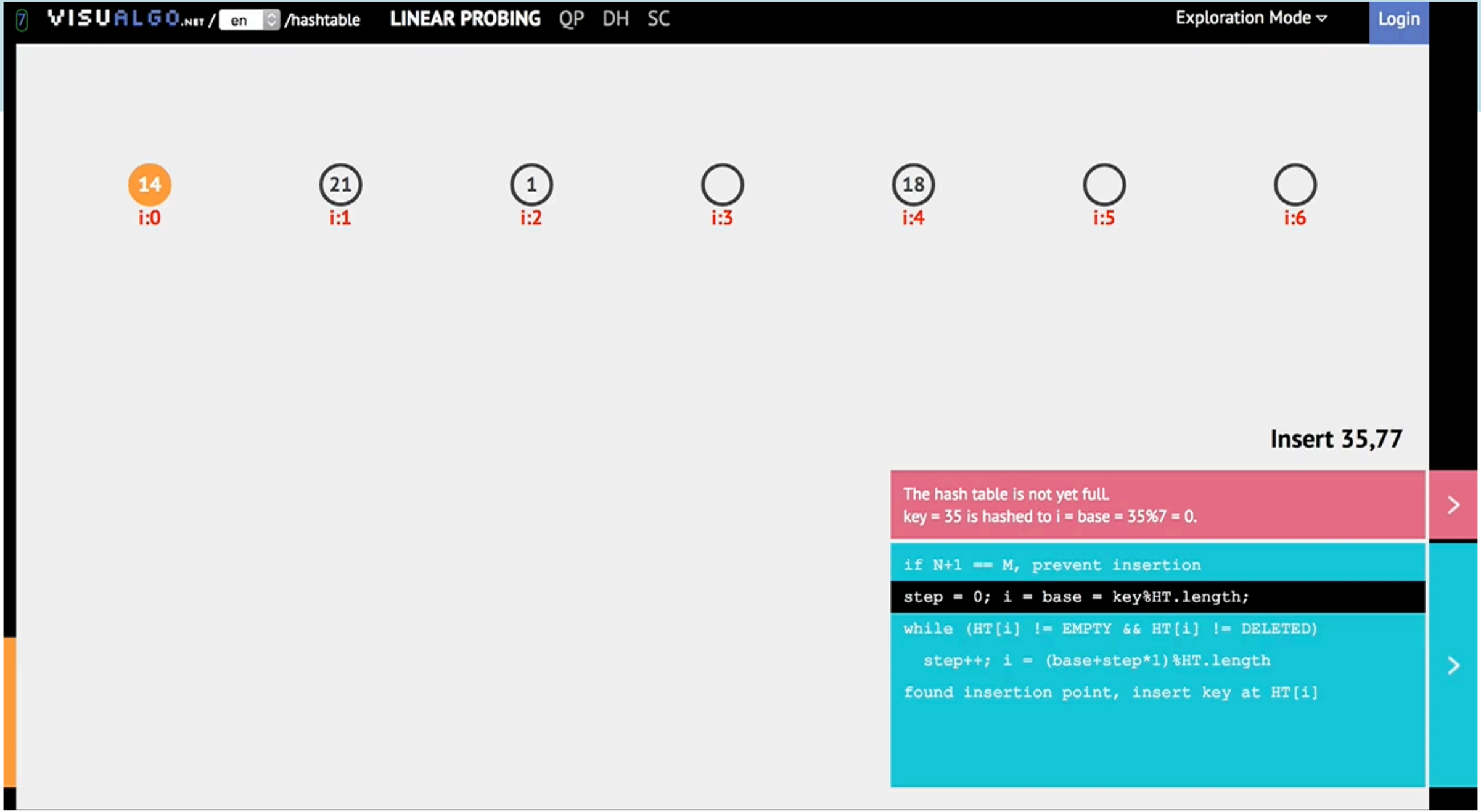
Each table cell inspected is referred to as a "probe"

Colliding items lump together, causing future collisions to cause a longer sequence of probes



`h(k) = k mod 11`


VISUALGO.NET / en /hashtable
LINEAR PROBING
QP
DH
SC
Exploration Mode
Login
14
i:0
21
i:1
1
i:2
i:3
18
i:4
i:5
i:6
Insert 35,77
The hash table is not yet full.
key = 35 is hashed to i = base = 35%7 = 0.
if N+1 == M, prevent insertion
step = 0; i = base = key%HT.length;
while (HT[i] != EMPTY && HT[i] != DELETED)
step++; i = (base+step*1)%HT.length
found insertion point, insert key at HT[i]


VISUALGO.NET / en /hashtable
LINEAR PROBING
QP
DH
SC
Exploration Mode
Login
14
i:0
21
i:1
1
i:2
i:3
18
i:4
i:5
i:6
Insert 35,77
The hash table is not yet full.
key = 35 is hashed to i = base = 35%7 = 0.
if N+1 == M, prevent insertion
step = 0; i = base = key%HT.length;
while (HT[i] != EMPTY && HT[i] != DELETED)
step++; i = (base+step*1)%HT.length
found insertion point, insert key at HT[i]

Linear Probing

if we try to insert an entry (k, v) into a bucket $A[j]$ that is already occupied, where $j = h(k)$, then we next try $A[(j+1) \bmod N]$.

If $A[(j+1) \bmod N]$ is also occupied, then we try $A[(j + 2) \bmod N]$, and so on, until we find an empty bucket that can accept the new entry.

Once this bucket is located, we simply insert the entry there.

This collision resolution strategy requires that we change the implementation when searching for an existing key—the first step of all get, put, or remove operations.

In particular, to attempt to locate an entry with key equal to k, we must examine consecutive slots, starting from $A[h(k)]$, until we either find an entry with an equal key or we find an empty bucket.

»

Linear Probing

To implement a deletion, we cannot simply remove a found entry from its slot in the array.

A way to get around this difficulty is to replace a deleted entry with a special "defunct" sentinel object.

With this special marker possibly occupying spaces in our hash table, we modify our search algorithm so that the search for a key k will skip over cells containing the defunct sentinel and continue probing until reaching the desired entry or an empty bucket (or returning back to where we started from).

Additionally, our algorithm for put should remember a defunct location encountered during the search for k, since this is a valid place to put a new entry (k,v), if no existing entry is found beyond it.

Linear Probing

Consider a hash table A that uses linear probing

get(k)

- We start at cell h(k)
- We probe consecutive locations until one of the following occurs
- An item with key k is found, or
- An empty cell is found, or
- N cells have been unsuccessfully probed

```
1:  function GET(k)
2:      i ← h(k)
3:      p ← 0
4:      do
5:          c ← A[i]
6:          if c = ∅ then
7:              return ∅
8:          else if c.key() = k then
9:              return c.getElement()
10:         else
11:             i ← (i+1) mod N
12:             p ← p + 1
13:         end if
14:     while p = N
15:     return ∅
16: end function
```

Linear Probing

To handle insertions and deletions, we introduce a special object, called AVAILABLE, which replaces deleted elements

remove(k)

- We search for an entry with key k
- If such an entry (k, o) is found, we replace it with the special item AVAILABLE and we return element o
- Else, we return null

put(k, o)

- We throw an exception if the table is full
- We start at cell h(k)
- We probe consecutive cells until one of the following occurs
- A cell i is found that is either empty or stores AVAILABLE, or N cells have been unsuccessfully probed
- We store entry (k, o) in cell i

Hashing Performance

In the worst case, searches, insertions and removals on a hash table take $O(n)$ time

The worst case occurs when all the keys inserted into the map collide

The load factor $\lambda = n/N$ affects the performance of a hash table

Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

$$1 / (1 - \lambda)$$

The expected running time of all operations in a hash table is $O(1)$

In practice, hashing is very fast provided the load factor is not close to 100%

Set

A *set* is an unordered collection of elements, without duplicates

typically supports efficient membership tests

no keys and elements are in no particular order

Set ADT

add(e)	Adds the element e to S
contains(e)	returns true if the element e is in S, otherwise false
remove(k)	removes the element e from S
iterator()	returns an iterators of the elements in S

Java Map

java.util
Interface Map<K,V>

- Type Parameters:
  K - the type of keys maintained by this map
  V - the type of mapped values

  All Known Subinterfaces:
  Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

  All Known Implementing Classes:
  AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Java Map

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map (optional operation).
boolean	**containsKey(Object key)** Returns true if this map contains a mapping for the specified key.
boolean	**containsValue(Object value)** Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
V	**get(Object key)** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put(K key, V value)** Associates the specified value with the specified key in this map (optional operation).
V	**remove(Object key)** Removes the mapping for a key from this map if it is present (optional operation).
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a Collection view of the values contained in this map.

Java Hashtable

java.util
Class Hashtable<K,V>

- java.lang.Object
  - java.util.Dictionary<K,V>
    - java.util.Hashtable<K,V>
- All Implemented Interfaces:
  Serializable, Cloneable, Map<K,V>

This example creates a hashtable of numbers. It uses the names of the numbers as keys:

```
Hashtable<String, Integer> numbers = new Hashtable<String, Integer>();
numbers.put("one", 1);
numbers.put("two", 2);
numbers.put("three", 3);
```

To retrieve a number, use the following code:

```
Integer n = numbers.get("two");
if (n != null) {
  System.out.println("two = " + n);
}
```

Hashtable or Hashmap?

Insertion Order : Neither HashMap nor Hashtable guarantee that the order of the map. If you need to guarantee the order use LinkedHashMap

2. Map interface : Both HashMap and Hashtable implement Map interface.

3. Put and get method : Both HashMap and Hashtable provide constant time performance for put and get methods assuming that the objects are distributed uniformly across the bucket.

4. Internal working : Both HashMap and Hashtable works on the Principle of Hashing.

Hashtable or Hashmap?

1. **Synchronisation or Thread Safe :** HashMap is not synchronised, but HashTable is synchronised

2. **Null keys and null values :** Hashmap allows one null key and any number of null values, while Hashtable does not allow null keys and null values in the HashTable object.

3. **Iterating the values:** Hashmap object values are iterated by using iterator. HashTable is the only class other than vector which uses enumerator to iterate the values of HashTable object.

4. **Fail-fast iterator** : The iterator in Hashmap is fail-fast iterator while the enumerator for Hashtable is not.

5. **Performance :** Hashmap is much faster and uses less memory than Hashtable as former is unsynchronised.

`Prefer HashMap!`

Java Hashcode

- The hash code function for String class in Java, where terms are summed using Java 32-bit `int` addition, s[i] denotes the UTF-16 code unit of the i'th character of the string, and n is the length of `s`

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

character	UTF16
A	65
B	66
C	67
D	68
E	69
...	...

This is a polynomial accumulation method.

The multiplier of this function is 31 and it has its own rationale

Joshua Bloch in Effective Java explains: "The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance: 31 * i == (i << 5) - i. Modern VMs do this sort of optimisation automatically."

java.lang.String (jdk1.8)

```
/**
 * Returns a hash code for this string. The hash code for a
 * {@code String} object is computed as
 * s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
 * using integer arithmetic, where s[i]e is the
 * i'th character of the string, n is the length of
 * the string, and ^ indicates exponentiation.
 * (The hash value of the empty string is zero.)
 */
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

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

Collisions

Number of **32-bit** hash values	Number of **64-bit** hash values	Number of **160-bit** hash values	Odds of a hash collision
77163	5.06 billion	1.42×10^{24}	1 in 2
30084	1.97 billion	5.55×10^{23}	1 in 10
9292	609 million	1.71×10^{23}	1 in 100
2932	192 million	5.41×10^{22}	1 in 1000
927	60.7 million	1.71×10^{22}	1 in 10000
294	19.2 million	5.41×10^{21}	1 in 100000
93	6.07 million	1.71×10^{21}	1 in a million
30	1.92 million	5.41×10^{20}	1 in 10 million
10	607401	1.71×10^{20}	1 in 100 million
	192077	5.41×10^{19}	1 in a billion
	60740	1.71×10^{19}	1 in 10 billion
	19208	5.41×10^{18}	1 in 100 billion
	6074	1.71×10^{18}	1 in a trillion
	1921	5.41×10^{17}	1 in 10 trillion
	608	1.71×10^{17}	1 in 100 trillion
	193	5.41×10^{16}	1 in 10^{15}
	61	1.71×10^{16}	1 in 10^{16}
	20	5.41×10^{15}	1 in 10^{17}
	7	1.71×10^{15}	1 in 10^{18}

Odds of a full house in poker
1 in 693

Odds of four-of-a-kind in poker
1 in 4164

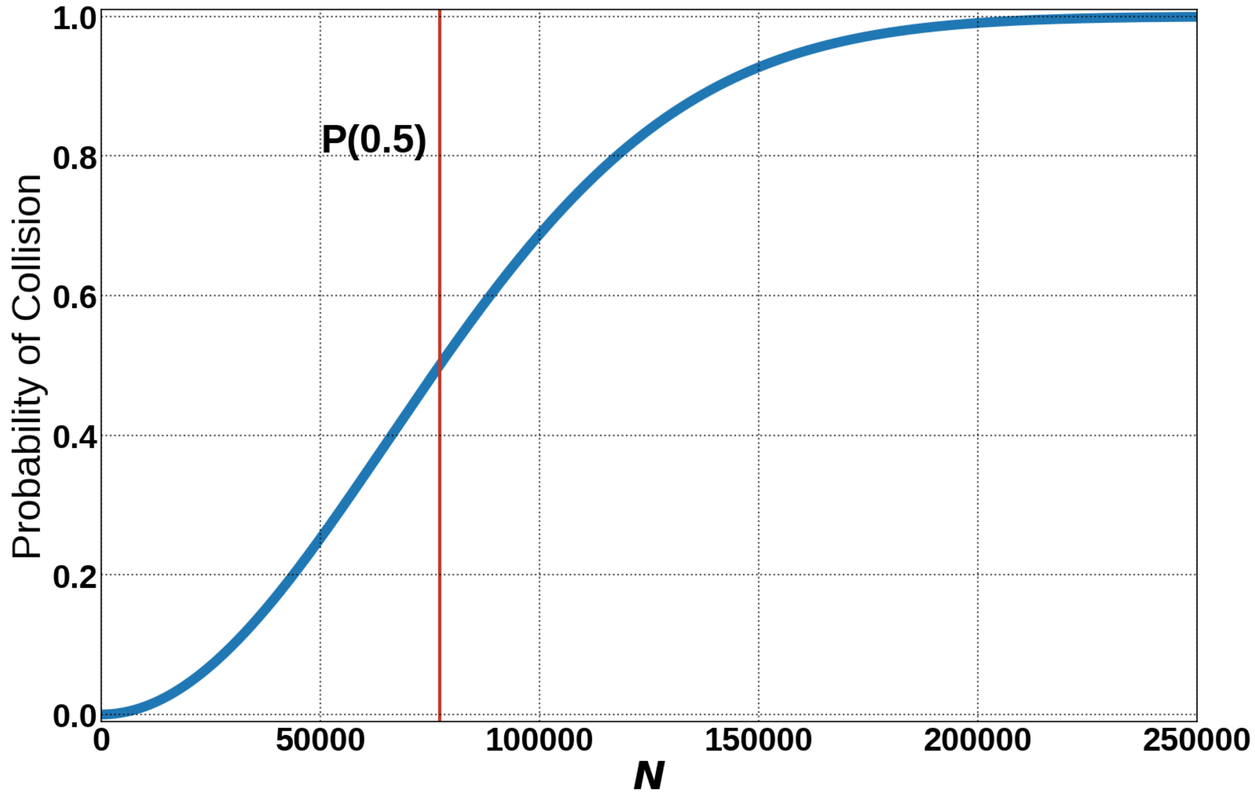
Odds of being struck by lightning
1 in 576000

Odds of winning a 6/49 lottery
1 in 13.9 million

Odds of dying in a shark attack
1 in 300 million

Odds of a meteor landing on your house
1 in 182 trillion

32-bit hash table



with integer keys and small number of values, the probability of a collision is very small. For a hash table size of >77163 the probability of a collision is 1 in 2.

Collisions

Number of **32-bit** hash values	Number of **64-bit** hash values	Number of **160-bit** hash values	Odds of a hash collision
77163	5.06 billion	1.42×10^{24}	1 in 2
30084	1.97 billion	5.55×10^{23}	1 in 10
9292	609 million	1.71×10^{23}	1 in 100
2932	192 million	5.41×10^{22}	1 in 1000
927	60.7 million	1.71×10^{22}	1 in 10000
294	19.2 million	5.41×10^{21}	1 in 100000
93	6.07 million	1.71×10^{21}	1 in a million
30	1.92 million	5.41×10^{20}	1 in 10 million
10	607401	1.71×10^{20}	1 in 100 million
	192077	5.41×10^{19}	1 in a billion
	60740	1.71×10^{19}	1 in 10 billion
	19208	5.41×10^{18}	1 in 100 billion
	6074	1.71×10^{18}	1 in a trillion
	1921	5.41×10^{17}	1 in 10 trillion
	608	1.71×10^{17}	1 in 100 trillion
	193	5.41×10^{16}	1 in 10^{15}
	61	1.71×10^{16}	1 in 10^{16}
	20	5.41×10^{15}	1 in 10^{17}
	7	1.71×10^{15}	1 in 10^{18}

Odds of a full house in poker
1 in 693

Odds of four-of-a-kind in poker
1 in 4164

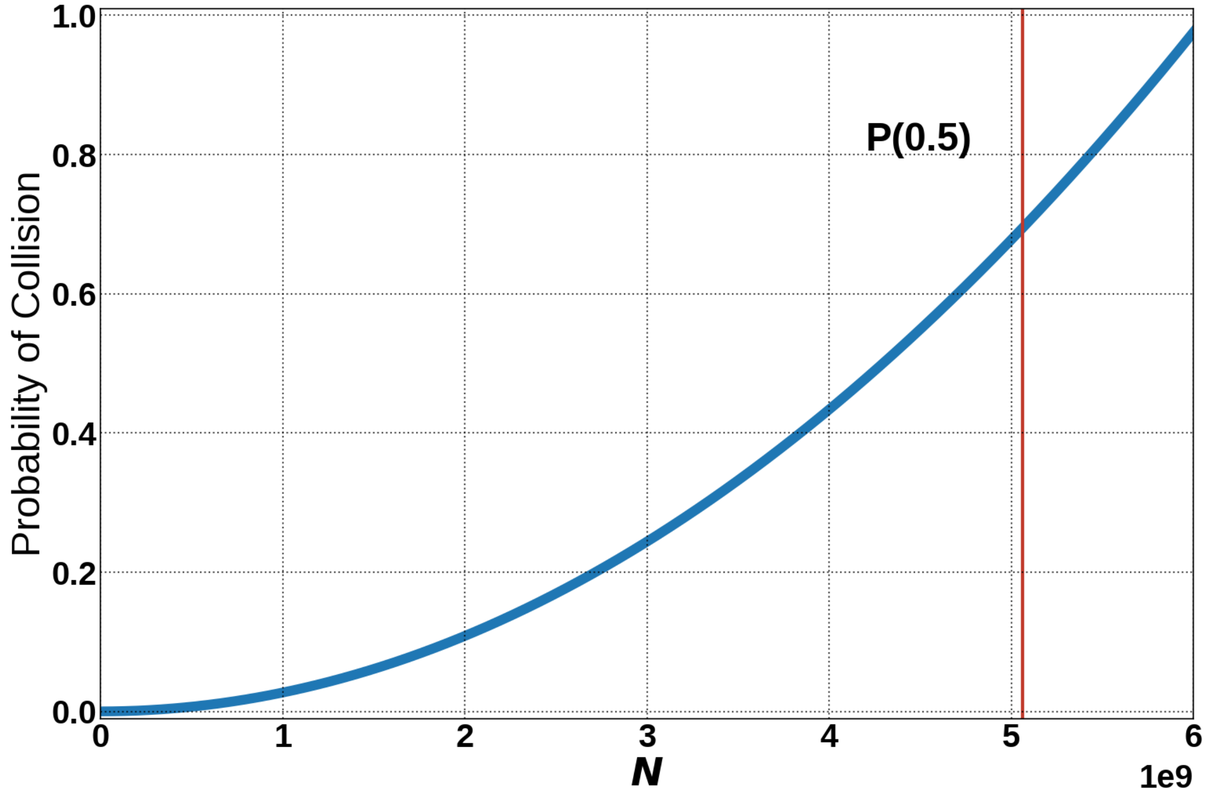
Odds of being struck by lightning
1 in 576000

Odds of winning a 6/49 lottery
1 in 13.9 million

Odds of dying in a shark attack
1 in 300 million

Odds of a meteor landing on your house
1 in 182 trillion

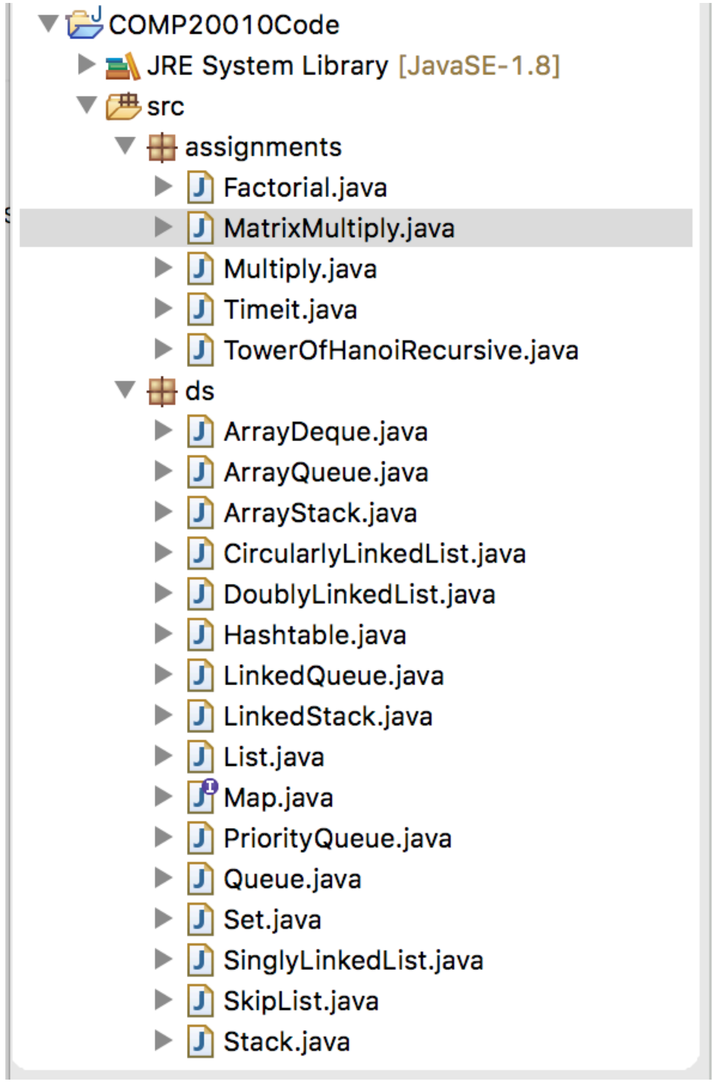
64-bit hash table



with long keys and small number of values, the probability of a collision is very small. For a hash table size of >5.06billion the probability of a collision is 1 in 2.

Code Repository

Code Repository



Should contain all your data structures implementations

and code for practicals/ tutorials/assignments

Zip the project for submission

Use comments to document the code

Implement a main method to test your classes.

Pay attention to the organisation/structure

- List
  - SinglyLinkedList
  - DoublyLinkedList
  - CircularlyLinkedList
- Stack
  - ArrayStack
  - LinkedListStack
- Queue
  - ArrayQueue
  - LinkedListQueue
- PriorityQueue

What's Next

- Review of material
- About the exam
- Any questions