FALL 2018 CISC 311

OBJECT & STRUCTURE & ALGORITHM I

– Chapter 9: Maps and Hash Tables

Outline

- The map ADT
- Hash table
- Implementations

The Map ADT

- 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

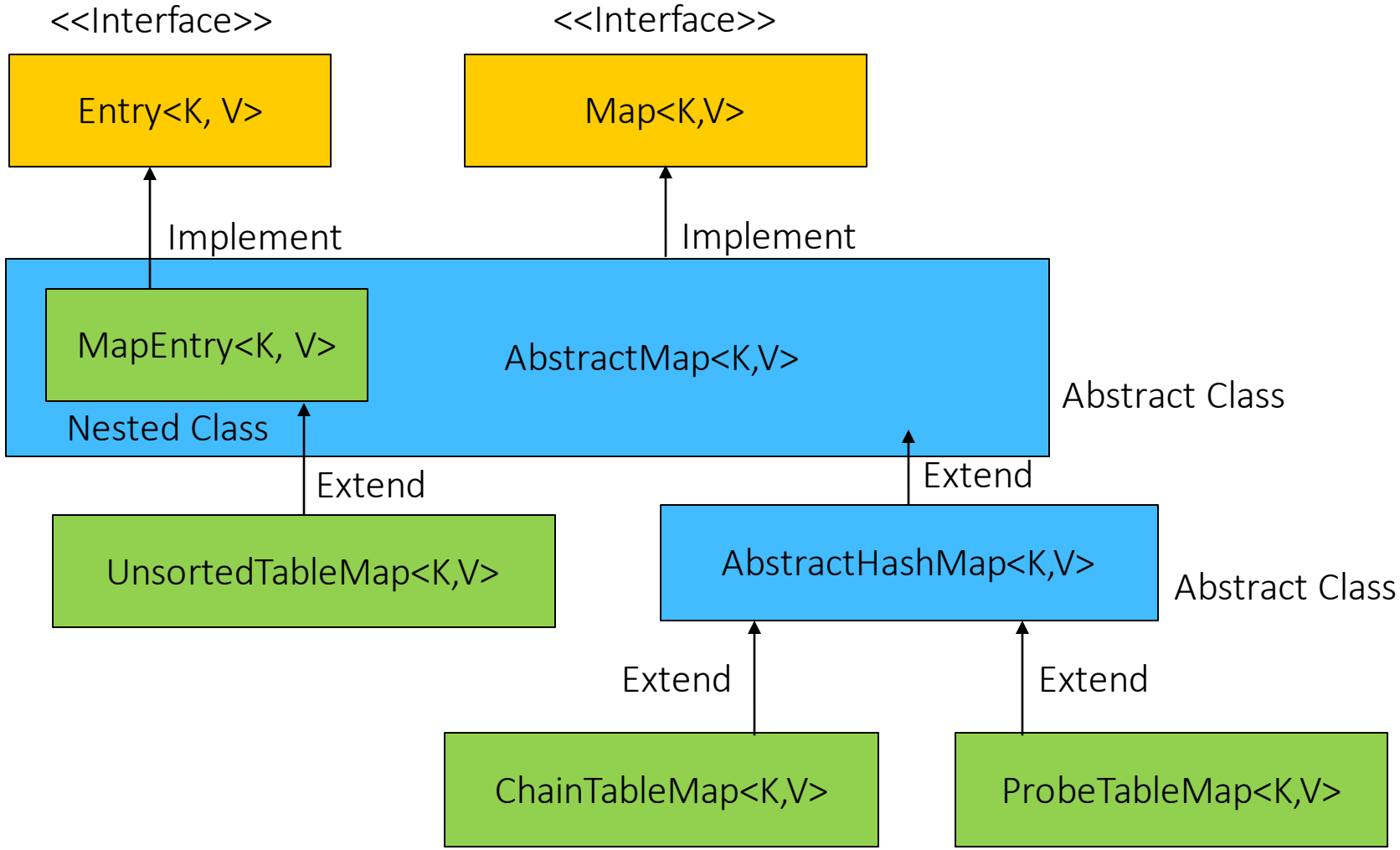
Map Methods

Method	Description
`size()`	Returns the number of elements in the map M
`isEmpty()`	Returns a boolean indicating whether the map M is empty
`get(k)`	If the map M has an entry with key k, returns its associated value v; else, returns `null`
`put(k,v)`	If the map M does not have an entry with key k, then adds entry (k, v) into the map M and returns `null`; else, replaces the value associated with k with v and returns the old value associate 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`

Map Methods (cont'd.)

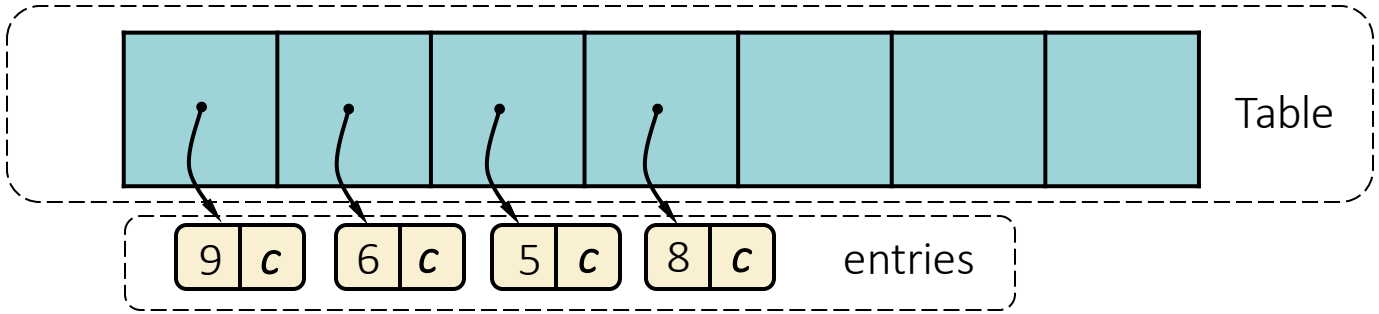
Method	Description
`entrySet()`	Returns an iterable collection of the entries in *M*
`keySet()`	Returns an iterable collection of the keys in *M*
`values()`	Returns an iterator of the values in *M*

Implementation



A Simple List-Based Map

- We can implement a map using an unsorted list
  - We store the items of the map in a list *S* (based on an array list), in arbitrary order



UnsortedTableMap.java

Performance of a List-Based Map

- Performance:
  - `put()` takes $O(n)$ time since we need to scan through the entire list when search for an existing entry, even though the insertion only takes $O(1)$ time
  - `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 (e.g., historical record of logins to a workstation)

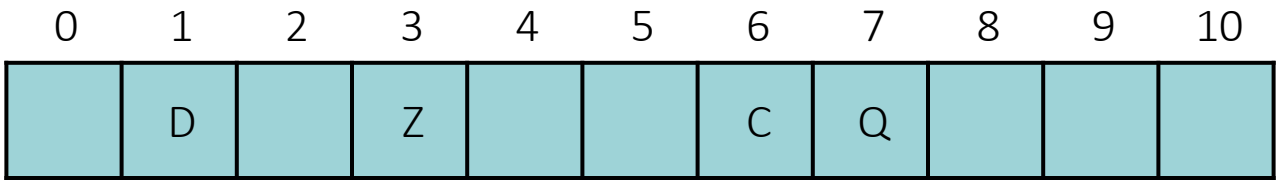
Performance of a List-Based Map (cont'd.)

Method	Unsorted array list
`size()`	$O(1)$
`isEmpty()`	$O(1)$
`get(k)`	$O(n)$
`put(k,v)`	$O(n)$
`remove(k)`	$O(n)$

Not efficient

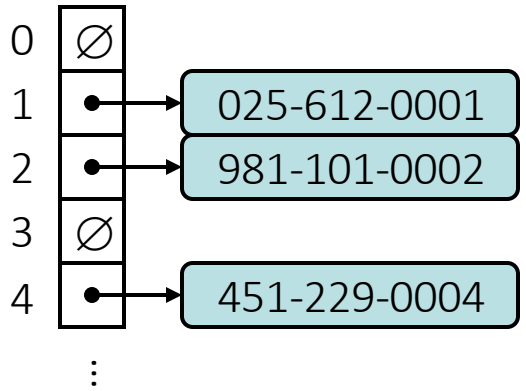
Intuitive Notion of a Map

- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as $M[k]$
- As a mental warm-up, 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 $N \geq n$



More General Kinds of Keys

- But what should we do if our keys are not integers in the range from 0 to $N - 1$?
  - Use a hash function to map general keys to corresponding indices in a table
  - For instance, the last four digits of a Social Security number



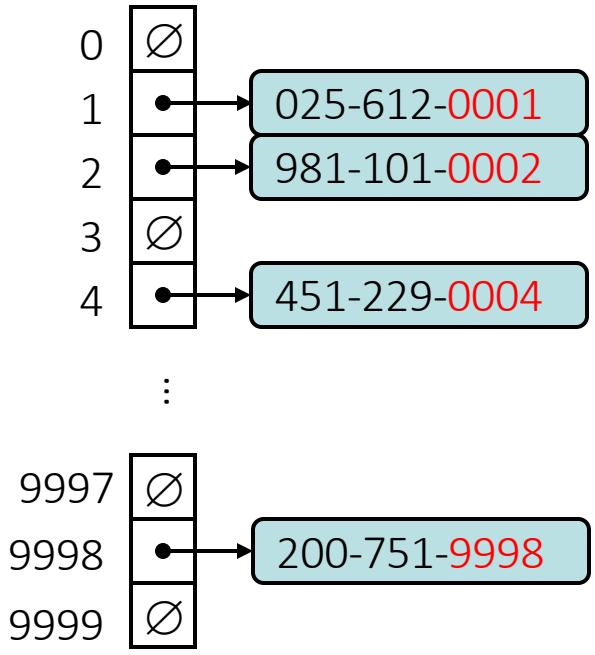
Hash Functions and Hash Tables

- 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
- The integer $h(x)$ 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 $i = h(k)$

Example

- 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 Functions

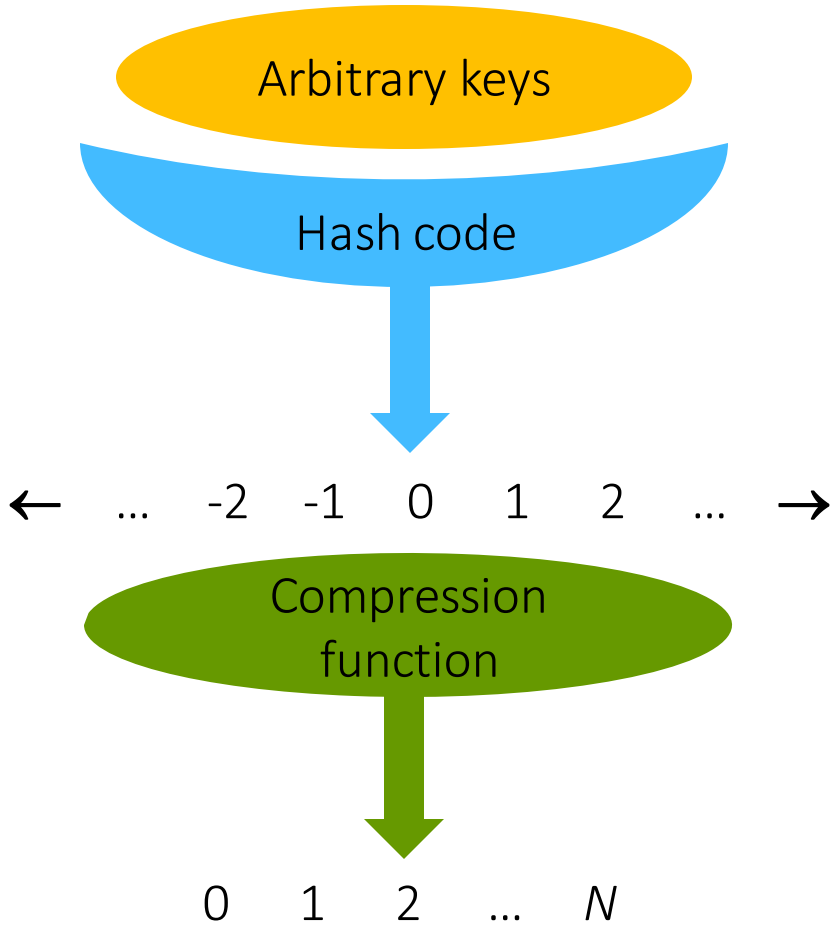
- Hash function h: mapping from *keys* to the slots of a hash table $T[0 \ldots N-1]$

$$h : keys \rightarrow \{0, 1, \ldots, N-1\}$$

- With arrays, key k maps to slot $A[k]$
- With hash tables, key k maps or "hashes" to slot $T[h[k]]$
- $h[k]$ is the hash value of key k

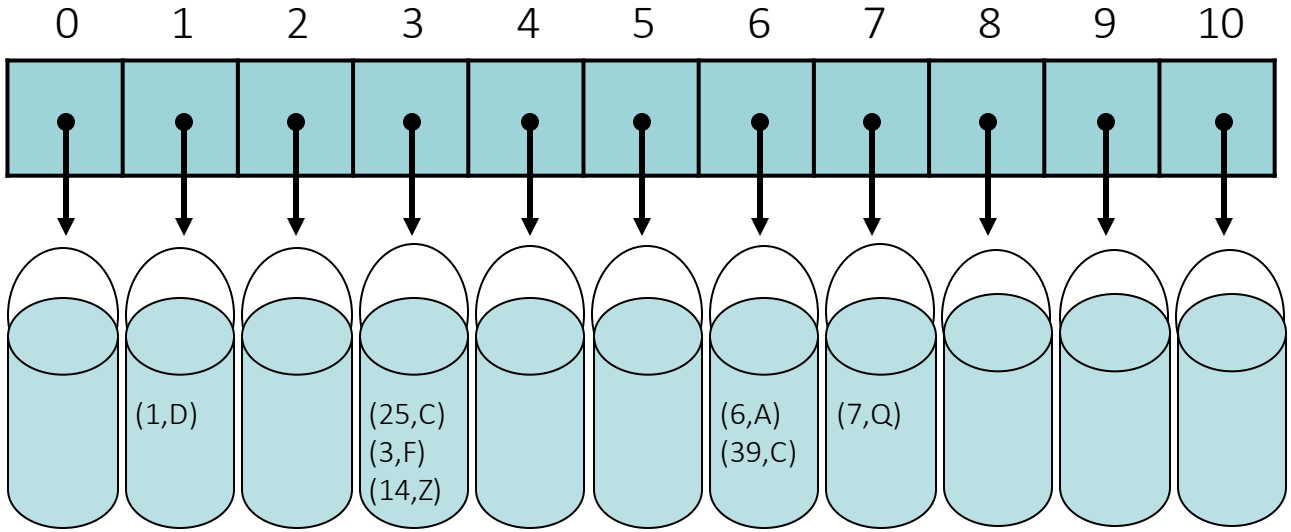
Hash Functions (cont'd.)

- A hash function is usually specified as the composition of two functions:
  - Hash code:
    h_1: keys $\rightarrow$ integers
  - Compression function:
    h_2: integers $\rightarrow [0, N - 1]$
- 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 Tables

- Ideally, keys will be well distributed in the range from 0 to $N - 1$ by a hash function, but in practice there may be two or more distinct keys that get mapped to the same index
- We conceptualized the hash table as a bucket array
  - Every bucket manage a collection of entries that are sent to a specific index by the hash function
  - Entry (k, v) is stored in bucket $A[h(k)]$



Choosing a Hash Function

- Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key
- A good hash function and implementation algorithm are essential for good hash table performance, but may be difficult to achieve
- If all keys are known ahead of time, a perfect hash function can be used to create a perfect hash table that has no collisions. If minimal perfect hashing is used, every location in the hash table can be used as well
- Such collisions must be accommodated in some way

Choosing a Hash Function (cont'd.)

- Let's say we want to store a 50,000-word English-language dictionary in main memory
  - You would like every word to occupy its own cell in a 50,000-cell array, so you can access the word using an index number. This will make access very fast
  - But what's the relationship of these index numbers to the words?
- Converting Words to Numbers
  - E.g., ASCII code

Choosing a Hash Function (cont'd.)

- Adding the digits: a simple approach to convert a word to a number might be to simply add the code numbers for each character
  - E.g., convert the word cats to a number
  - First, we convert the characters to digits using our homemade code: c = 3 a = 1 t = 20 s = 19
  - Then we add them: 3 + 1 + 20 + 19 = 43
  - Thus, in our dictionary the word cats would be stored in the array cell with index 43
- Problem: all the other English words would likewise be assigned an array index calculated by this process

Choosing a Hash Function (cont'd.)

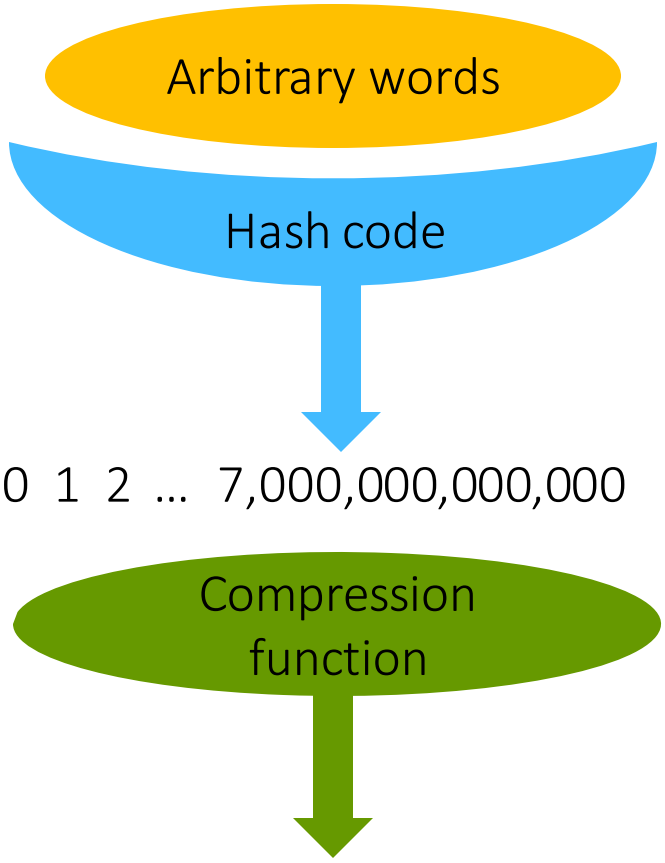
- Let's restrict ourselves to 10 letter words. Then (remembering that a blank is 0),
  - The first word in the dictionary a, would be coded by 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1
  - The last potential word in the dictionary would be zzzzzzzzzz (10 Zs). Our code obtained by adding its letters would be 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 = 260
- Each bucket need to hold 50000 / 260 = 192 words
- Too many words have the same index → Accessing the array element would be quick, but searching through the 192 words to find the one we wanted would be slow

Choosing a Hash Function (cont'd.)

- Multiplying by Powers: an approach that creates an array in which every word, in fact every potential word, from *a* to *zzzzzzzzzz*, was guaranteed to occupy its own unique array element
  - Decompose a word into its letters, convert the letters to their numerical equivalents, multiply them by appropriate powers of 27 (because there are 27 possible characters, including the blank), and add the results. This gives a unique number for every word
  - E.g., convert the word cats to a number
    - Index $= 3 \times 27^3 + 1 \times 27^2 + 20 \times 27^1 + 19 \times 27^0 = 60337$
    - Thus, in our dictionary the word cats would be stored in the array cell with index 60337
- The problem is that this scheme assigns an array element to every potential word, whether it's an actual English word or not

Choosing a Hash Function (cont'd.)

- Compress the huge range of numbers into a range that matches a reasonably sized array



Hash Codes

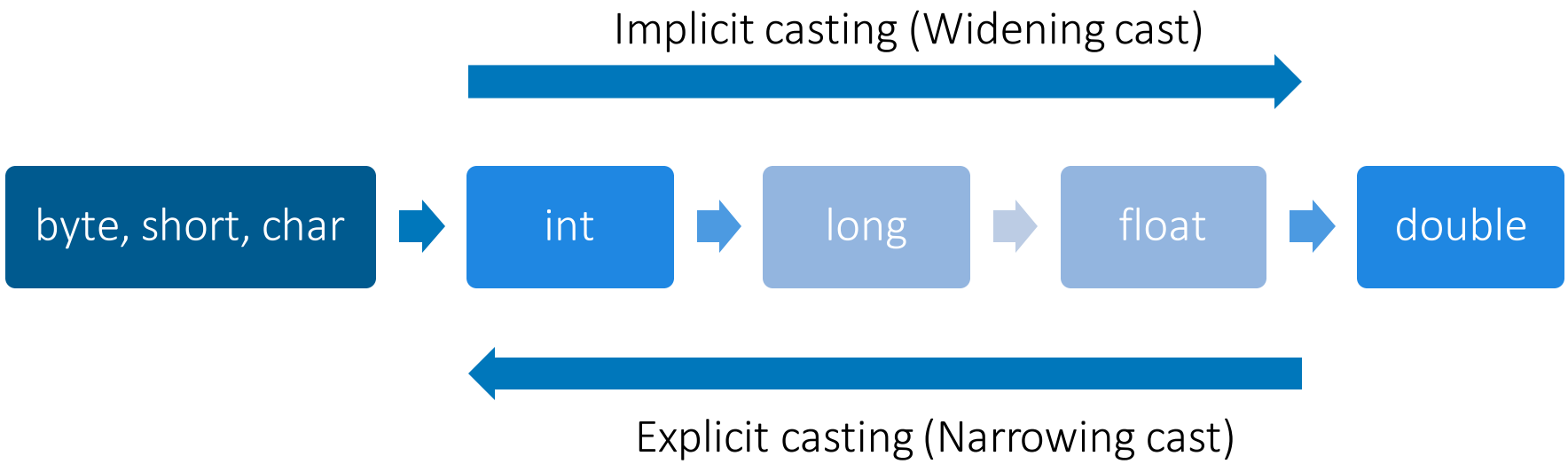
- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal 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)

Recall Primitive Types

- Java has several base types, which are basic ways of storing data
- An identifier variable can be declared to hold any base type and it can later be reassigned to hold another value of the same type

	bits	bytes	values
`byte`	8	1	$-2^7 \sim 2^7-1$
`short`	16	2	$-2^{15} \sim 2^{15}-1$
`int`	32	4	$-2^{31} \sim 2^{31}-1$
`long`	64	8	$-2^{63} \sim 2^{63}-1$
`float`	32	4	+/- 3.4×10^{38} with 7 significant digits
`double`	64	8	+/- 1.7×10^{308} with 15 significant digits
`char`	16	2	$0 \sim 2^{16}$
`boolean`	8	1	true / false

Recall Casting



Hash Codes (cont'd.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits) $a_0 a_1 \ldots a_{n-1}$
  - We evaluate the polynomial $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$ at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

Hash Codes (cont'd.)

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time

$$p_0(z) = a_{n-1}$$

$$p_i(z) = a_{n-i-1} + zp_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$

- We have $p(z) = p_{n-1}(z)$

Compression Functions

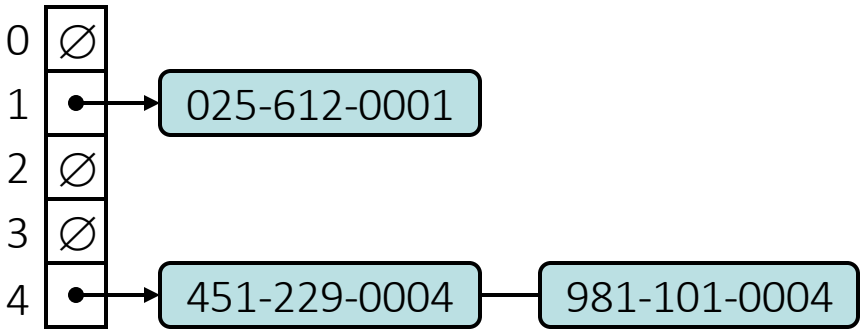
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = [(ay + b) \bmod p] \bmod N$
  - a: scale, b: shift
  - p is prime number larger than N
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Collision Handling

- Collisions occur when different elements are mapped to the same cell
- Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys
- For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is approximately a 95% chance of at least two of the keys being hashed to the same slot

Separate Chaining

- Separate Chaining: let each cell in the table point to a linked list of entries that map there
- Have each bucket $A[j]$ stores its own secondary container, holding all entries (k, v) such that $h(k) = j$
- Separate chaining is simple, but requires additional memory outside the table



Map with Separate Chaining

- Delegate operations to a list-based map at each cell:

```
Algorithm get(k)
    return A[h(k)].get(k)
```

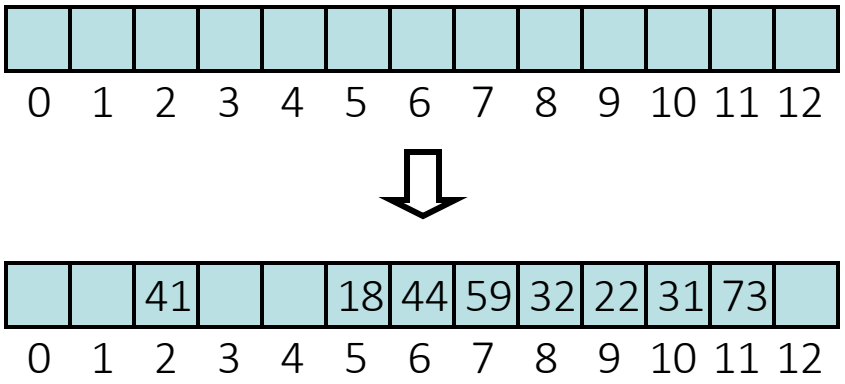
```
Algorithm put(k, v)
    t ← A[h(k)].put(k, v)
    if t = null then
        n ← n + 1
    return t
```

```
Algorithm remove(k)
    t ← A[h(k)].remove(k)
    if t ≠ null then
        n ← n – 1
    return t
```

ChainHashMap.java

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- 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
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Search with 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

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.getKey() = k
            return c.getValue()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

ProbeHashMap.java

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called `DEFUNCT`, 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 `DEFUNCT` 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 `DEFUNCT`, or
    - *N* cells have been unsuccessfully probed
  - We store (*k*, *o*) in cell *i*

Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

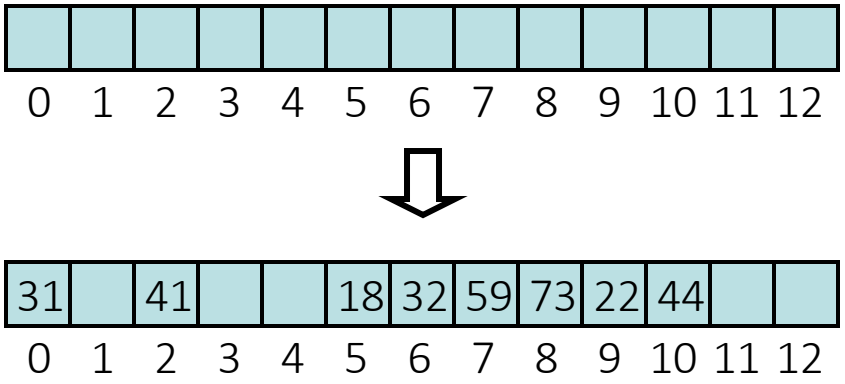
$$(i + jd(k)) \bmod N \text{ for } j = 0, 1, \ldots, N - 1$$

- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression function for the secondary hash function:
  - $d_2(k) = q - k \bmod q$
  - Where
    - $q < N$
    - q is a prime
- The possible values for $d_2(k)$ are $1, 2, \ldots, q$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Performance of Hashing

- 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 $1 / (1 - \lambda)$

Performance of Hashing

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - Small databases
  - Compilers
  - Browser caches

Performance of Hash Table

Method	Unsorted array list	Hash table	
		Expected	Worst case
`size()`	$O(1)$	$O(1)$	$O(1)$
`isEmpty()`	$O(1)$	$O(1)$	$O(1)$
`get(k)`	$O(n)$	$O(1)$	$O(n)$
`put(k,v)`	$O(n)$	$O(1)$	$O(n)$
`remove(k)`	$O(n)$	$O(1)$	$O(n)$

Summary

- The map ADT
  - Implementation
    - Unsorted list → not efficient
    - Hash table
    - Sorted list (not covered)
- The hash table ADT
  - Hash function
    - Hash code, compression function
  - Separate chaining
  - Open addressing
    - Linear probing
    - Double hashing