Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Fundamental Techniques In Handling People – Dale Carnegie

2. Give honest and sincere appreciation

Recap

- Elementary data structures
- ADT
- Array based vs. linked implementation
- Worst case time complexity to help us choose based on our needs

Today’s Objectives

- What is a “MAP or Dictionary ADT”?
- What choices do we have to implement a MAP?
- What is a hash function and a hash table?
- What is collision and how it handle it?
- How to analyze time complexity of a Hash Map?

Map or Dictionary

Map or Dictionary

- Models a searchable dynamic set of key-value entries

- Main operations are: *searching*, *inserting*, and *deleting* items

- Applications:

  - Compiler symbol table

  - A news indexing service

The Map ADT

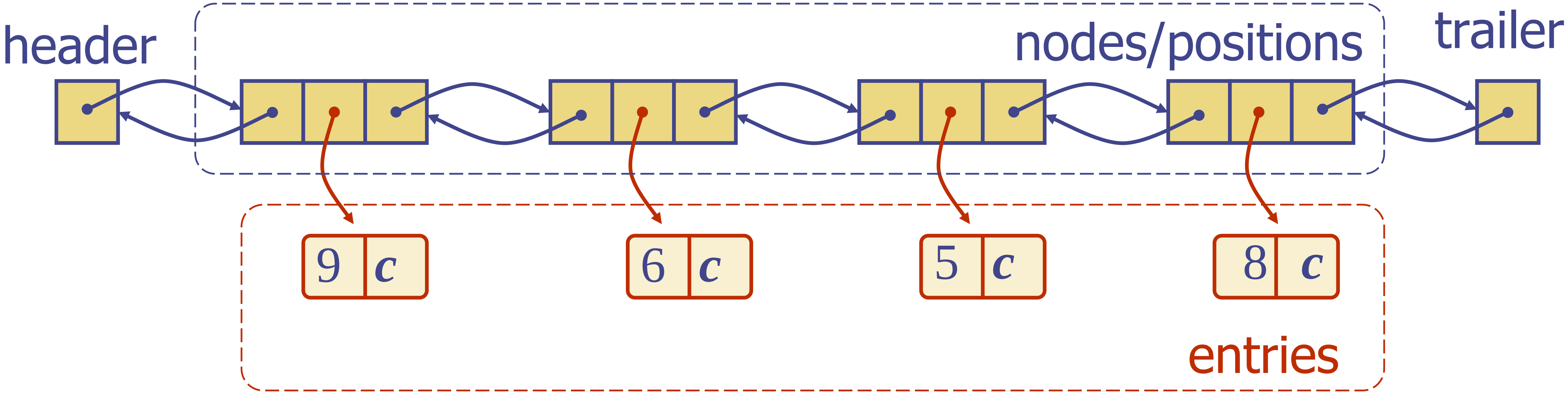
- **get(k):** if the map M has an entry with key k, return its associated value; else, return null
- **put(k, v):** insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- **remove(k):** if the map M has an entry with key k, remove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

Example

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)

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 a doublylinked list), in arbitrary order



The get(k) Algorithm

```
Algorithm get(k):
  while map.hasNext() do
    p = map.next()  { the next element in the map}
    if p.element().getKey() = k  then
        return p.element().getValue()
  return null {there is no entry with key equal to k}
```

The put(k,v) Algorithm

```
Algorithm put(k,v):
while map.hasNext() do
   p = map.next()
   if p.element().getKey() = k  then
     t = p.element().getValue()
     map.set(p,(k,v))
     return t   {return the old value}
map.addLast((k,v))
n = n + 1  {increment variable storing number of entries}
return null   { there was no entry with key equal to k }
```

The remove(k) Algorithm

```
Algorithm remove(k):
while map.hasNext() do
   p = map.next()
   if p.element().getKey() = k  then
     t = p.element().getValue()
     map.remove(p)
     n = n – 1     {decrement number of entries}
     return t {return the removed value}
return null     {there is no entry with key equal to k}
```

Performance of a List-Based Map

- Performance:
    - put takes $\boldsymbol{O}(1)$ time since we can insert the new item at the beginning or at the end of the sequence
    - get and remove take $\boldsymbol{O}(\boldsymbol{n})$ 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)

Hash Map

Let’s Start With this Question

- How much time does it take to lookup an item in an array, if you already know its index?

Example

- Suppose you’re writing a program to access employee records for a company with 1000 employees.
- Goal: **fastest possible access to any individual record**
- Each employee has a number from 1(founder) to 1000 (the most recent worker)
- Employees are seldom laid off, and even when they are, their record stays in the database.

Example (cont.)

- The easiest way to do this is by using an array (we already know the size)
- Each employee record occupies one cell of the array
- The index number of the cell is the employee number

```
empRecord rec = databaseArray[72];

databaseArray[totalEmployees++] = newRecord;
```

Example (cont.)

- The speed and simplicity of data access using this array-based database make it very attractive.
- However, it works in our example only because keys are well organized
  - Sequentially from 1 to a known maximum
  - No deletions required
  - New items can be added sequentially at the end

Example (cont.)

- But mostly, the keys are not so well behaved
- A simple example would be when keys are of type String.
- Array indexing requires integer
- One more problem: Even when using integers, the value could be outside of the range of the array

Hash Map

- Hash Table is a very practical way to maintain a map

- It consists of an **array** and **hash function**: a mathematical way of mapping an arbitrary key to an index in the array

Hash Table

- A **hash table** for a given key type consists of
  - ❖ Hash function h
  - ❖ Array (called table) of size N or m
- 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]

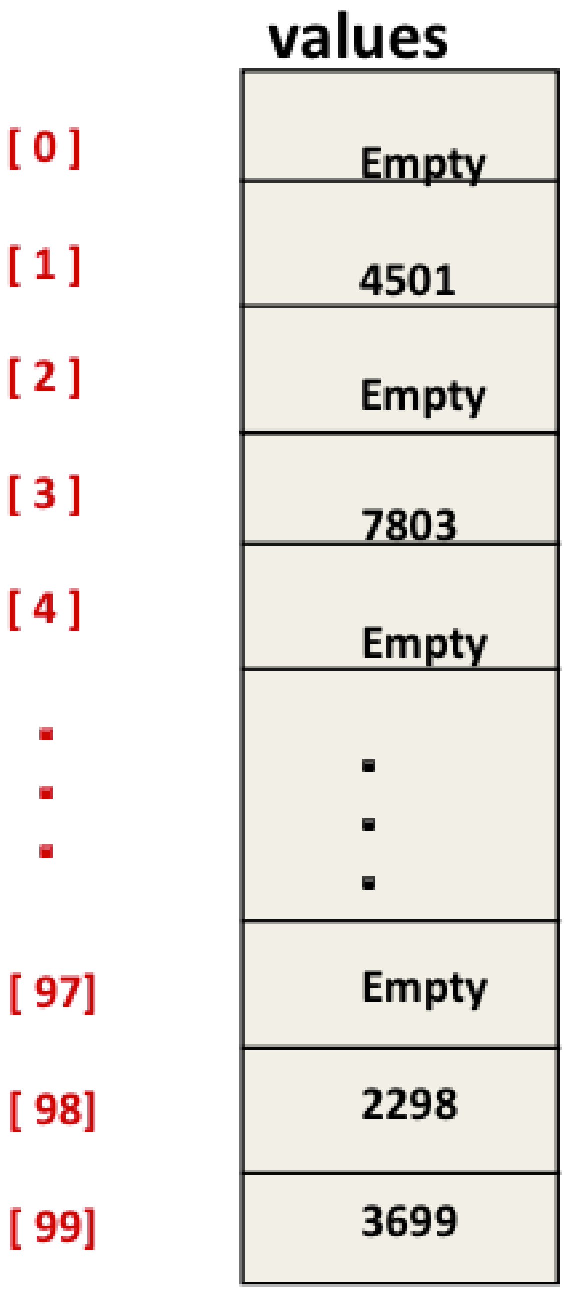
❖ Example:

h(x) = x mod N

is a hash function for integer keys

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

Hash Map



	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

HandyParts company
makes no more than 100
different parts. But the
parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 100

Hash Functions

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

Hash code:

$\boldsymbol{h}_1$: keys □ integers

Compression function:

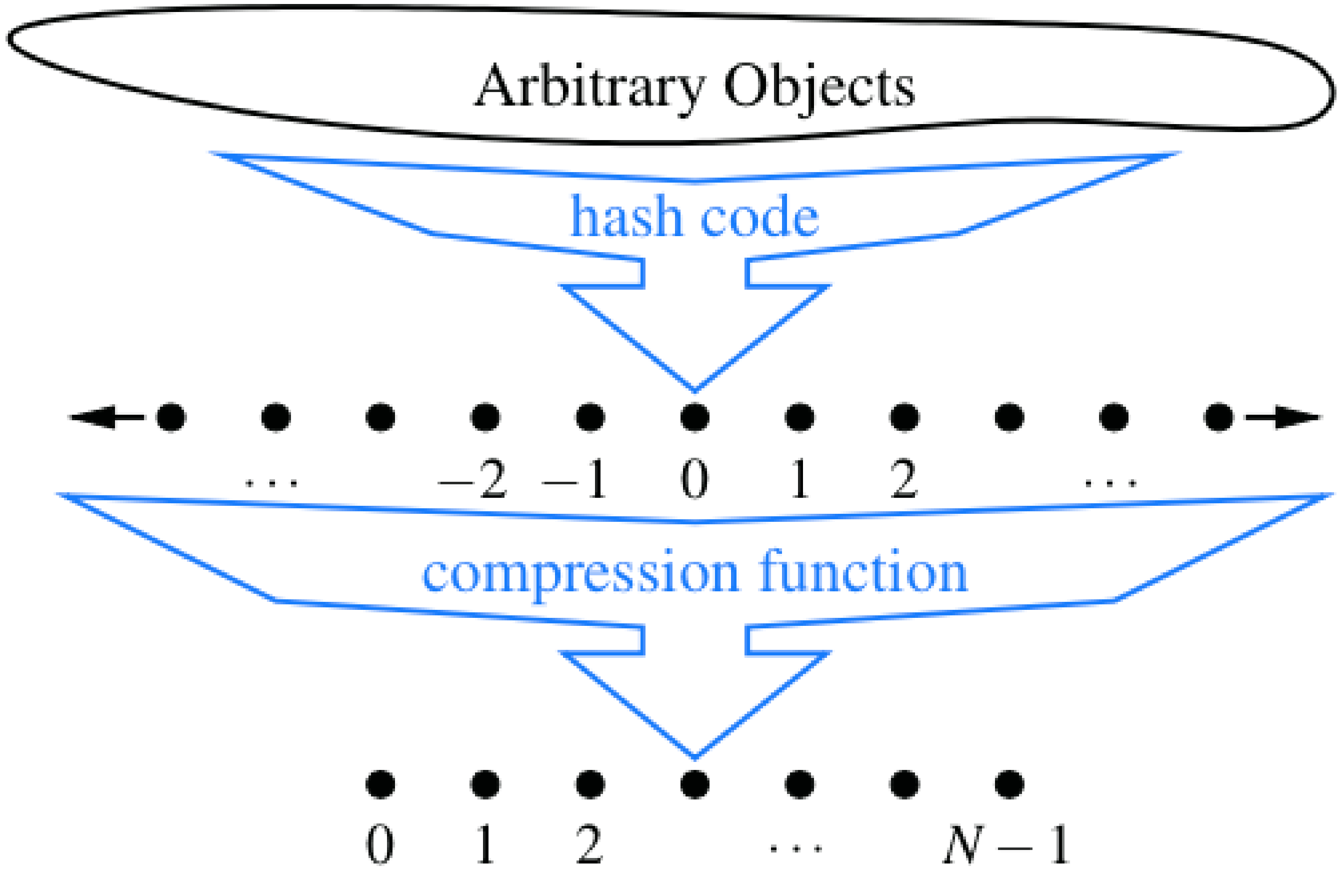
$\boldsymbol{h}_2$: integers □ $[0, \boldsymbol{N}\quad 1]$

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

  h(x) = h2(h1(x))

- The goal of the hash function is to “disperse” the keys in a uniform manner

Parts of a Hash Function



Ideal Hash Function

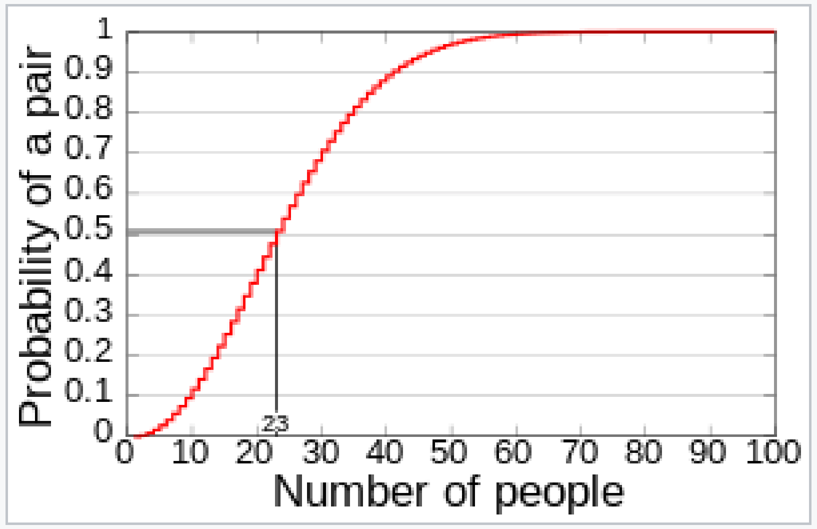
- Every resulting hash value has exactly one input that will produce it
- Same key hashes to the same index (**repeatable**)
- Hash value is widely different if even a single bit is different in the key (**avalanche**)
- Should work in general (for different types)

Some Principles

1. If n items are placed in m buckets, and n is greater than m, one or more buckets contain two or more items (Pigeonhole Principle)

   ❖ This is called collision (two keys hash to the same index)

2. Birthday paradox



The computed probability of at least two people sharing a birthday versus the number of people

https://en.wikipedia.org/wiki/Birthday_problem

Collisions

- So collisions are inevitable
- Our goal is to minimize collisions
- We will achieve it through:
    - Generating better hash codes
    - Performing better compression
    - Handling collisions

Tip!

- Designing a hash function is a black art
- It is always better to use a known good algorithm
- But sometimes, as a student, it is better to try to design one for the sake of practice

Hash Codes

1. **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 that it is not repeatable

Hash Codes (cont.)

2. **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)

Hash Codes (cont.)

3. **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

   - Fails to treat permutations differently (“abc”, “cba”, “cab”)

Hash Codes (cont.)

4. 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

$$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots$$
$$\ldots + a_{n\ \ 1} z^{n\ \ 1}$$

at a fixed value z

- Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

Compression Functions

1. **Division:**

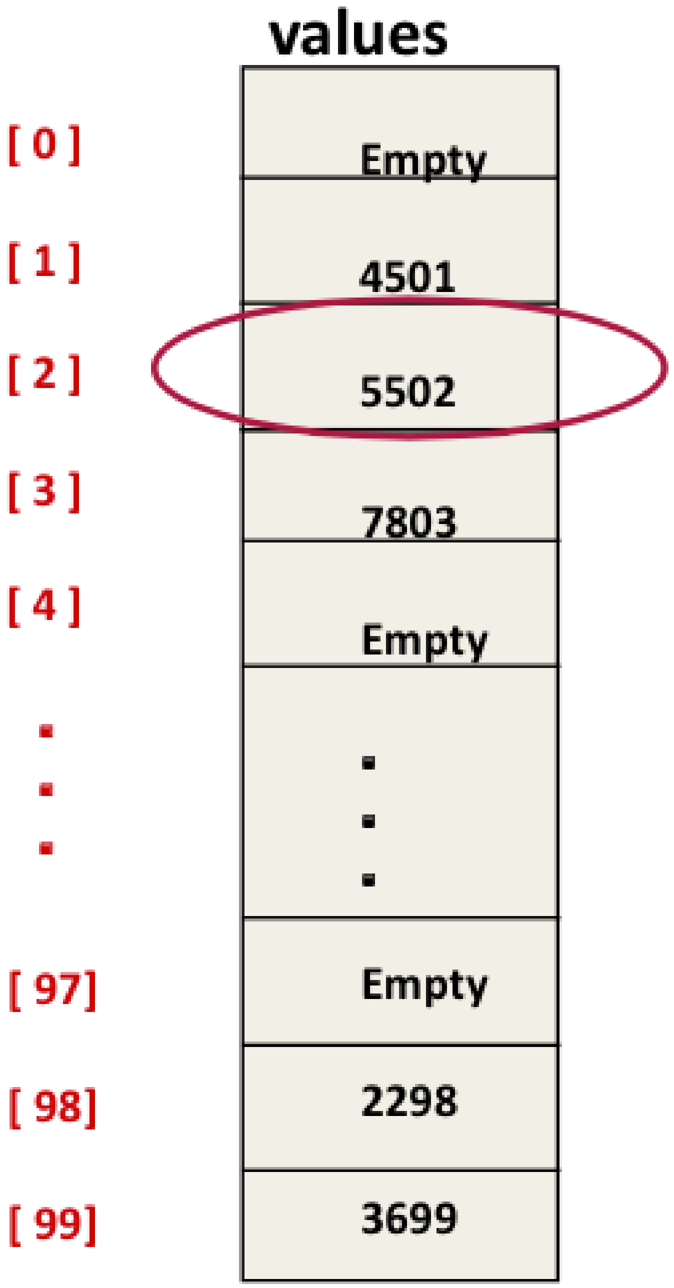
- $\boldsymbol{h}_2(\boldsymbol{y}) = \boldsymbol{y} \bmod \boldsymbol{N}$
- The size N of the hash table is usually chosen to be a prime
- Helps “spread out” the distribution of hashed values
- Try inset keys with hash codes {200, 205, 210, 215, …, 600} into a table size of 100 vs. 101

Compression Functions

2. Multiply, Add and Divide (MAD)

- $h_2(y) = [(ay + b) \bmod p] \bmod N$
- p is a prime number larger than N
- a and b are integers from the interval [0, p – 1], with a > 0

Collision



	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Next place part number 6702 in the array.

Hash(key) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

Collision Handling

- Let n be the number of items inserted into a hash table of size m

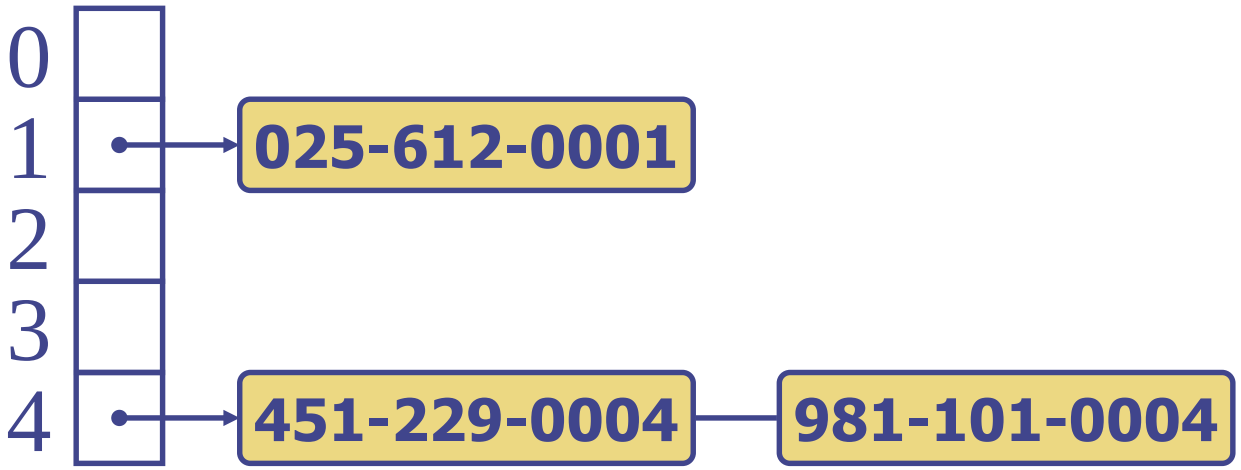
  - ❖ Separate Chaining: m much smaller than n

  - ❖ Open Addressing: m much larger than n

Collision Handling

1. Separate Chaining: let each cell in the table point to a linked list of entries that map there

- Separate chaining is simple, but requires additional memory outside the table



Analysis of get(k) in Separate Chaining

- Worst case: all elements get hashed to the same index or bucket, thus $O(n)$
- But if hash function is chosen well, the worst case is highly unlikely
- Thus we analyze the **expected** time complexity using the *load factor*

Analysis of get(k) in Separate Chaining

- Load factor (α): the ratio of n and m , represents the expected length of a chain

- The expected length of a chain in this case is $O(\alpha)$

- Thus, the expected time complexity in terms of the *load factor* - $O(1 + \alpha)$

- It is usually made sure that α doesn't exceed some constant, thus $O(1)$

Collision Handling

2. Open Addressing: the colliding item is placed in a different cell of the table

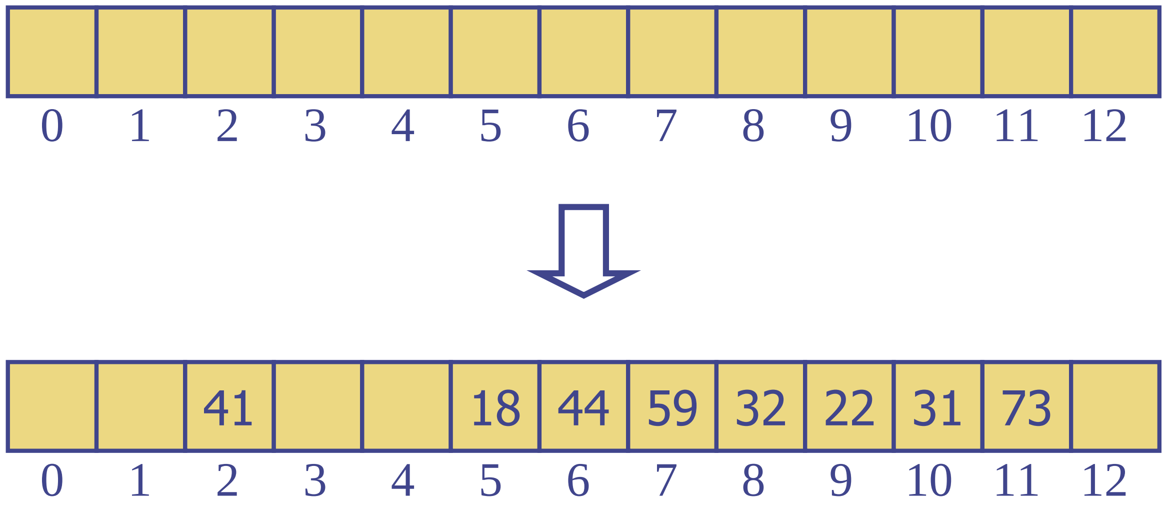
   A. Linear Probing: handles collision by placing the item in the next (circularly) available cell

      - Each cell inspected is called a probe

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

Example

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

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
```

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***

Updates with Linear Probing

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

Collision Handling

3. Open Addressing: the colliding item is placed in a different cell of the table

B. Double Hashing: uses a secondary hash function ***d(k)*** and handles collision by placing an items in the first available of cell of the series

$$(\boldsymbol{i} + \boldsymbol{j}\boldsymbol{d}(\boldsymbol{k})) \bmod \boldsymbol{N}$$

for $\boldsymbol{j} = 0,\ 1,\ \ldots\ ,\ \boldsymbol{N} \quad 1$

Double Hashing

- The secondary hash function cannot have zero values

- The table size ***N*** must be prime to allow probing of all the cells.

Double Hashing

- Common choice of compression function for the secondary hash function:

$$\boldsymbol{d}_2(\boldsymbol{k}) = \boldsymbol{q} \quad \boldsymbol{k} \bmod \boldsymbol{q}$$

where

$\boldsymbol{q} < \boldsymbol{N}$

$\boldsymbol{q}$ is a prime

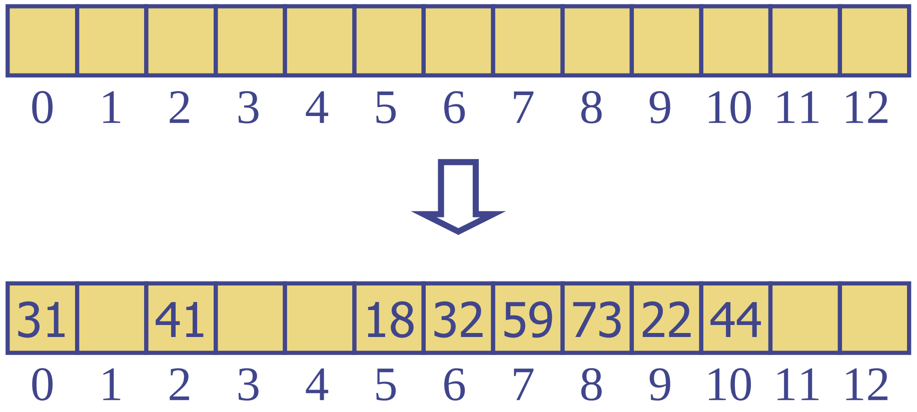
The possible values for $\boldsymbol{d}_2(\boldsymbol{k})$ are

$$1, 2, \ldots, \boldsymbol{q}$$

Example

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 \quad k \bmod 7$
- 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		



0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Analysis of get(k) in Open Addressing

- Load factor (α): the ratio of n and m

- The worst case time complexity in terms of the *load factor* - $O(\frac{1}{1-\alpha})$

- It is usually made sure that α doesn’t exceed some constant, thus $O(1)$

Did we achieve today’s objectives?

- What is a “MAP ADT”?
- What choices do we have to implement a MAP?
- What is a hash function and a hash table?
- What is collision and how it handle it?
- How to analyze time complexity of a Hash Map?