CSC 225 - Summer 2015

Hashing I

Bill Bird

Department of Computer Science
University of Victoria

June 26, 2015

The Pigeonhole Principle (1)

Exercise: Prove that in any group of 8 people, there must be at least two people whose birthdays will fall on the same day of the week in 2016.

The Pigeonhole Principle (2)

Exercise: Prove that in any group of 8 people, there must be at least two people whose birthdays will fall on the same day of the week in 2016.

Proof: Everyone in the group will have a birthday in 2016, and everyone's birthday must fall on one of the seven days of the week. Since there are only 7 days of the week, at least two birthdays must fall on the same day of the week.

The Pigeonhole Principle (3)

Exercise: Prove that in any group of 8 people, there must be at least two people whose birthdays will fall on the same day of the week in 2016.

Proof: Everyone in the group will have a birthday in 2016[a], and everyone's birthday must fall on one of the seven days of the week. Since there are only 7 days of the week, at least two birthdays must fall on the same day of the week.

[a]Since 2016 will be a leap year, this includes people born on February 29th.

The Pigeonhole Principle (4)

Exercise: Prove that in any group of 8 people, there must be at least two people whose birthdays will fall on the same day of the week in 2016.

- The proof is almost trivial, but it relies on an intuitive assumption: If any collection of n items is assigned to a set of m possible outcomes and $n > m$, at least two items must share an outcome.
- This is called the **Pigeonhole Principle**.

The Pigeonhole Principle (5)

Theorem: (The Pigeonhole Principle)
Suppose a group of n pigeons occupy a collection of m pigeonholes. If $m < n$, then at least one pigeonhole must contain multiple pigeons.

The Pigeonhole Principle (6)

Exercise: Consider a group of 1000 people with English names (both first and last). Prove that at least two people must have the same first and last initials.

The Pigeonhole Principle (7)

Exercise: Consider a group of 1000 people with English names (both first and last). Prove that at least two people must have the same first and last initials.

Since everyone has an English name, there are 26 possible choices for each of their initials. The total number of possible first/last initial pairs is

$$26 \cdot 26 = 676$$

and since $676 < 1000$, by the Pigeonhole Principle, two people in the group must have the same first and last initials.

The Pigeonhole Principle (8)

Exercise: Consider an array A of 114 distinct values in the range $\{0, 1, 2, \ldots, 225\}$. Prove that A must contain a pair of values x and y such that $x + y = 225$.

The Pigeonhole Principle (9)

Exercise: Consider an array A of 114 distinct values in the range $\{0, 1, 2, \dots, 225\}$. Prove that A must contain a pair of values x and y such that $x + y = 225$.

There are 113 possible pairs that add to 225:

$$(0, 225), \quad (1, 224), \quad \dots, \quad (111, 114), \quad (112, 113).$$

Each of the 114 elements of A falls into one of the pairs. Since $114 > 113$, by the Pigeonhole Principle there must be two elements $A[i]$ and $A[j]$ which fall into the same pair, so $A[i] + A[j] = 225$.

Tables (1)

Exercise: Describe a dictionary implementation which can store 5 distinct keys in the range $\{0, 1, 2, 3, \ldots, 10\}$.

Tables (2)

Index	Value
0	
1	1
2	
3	3
4	
5	
6	
7	
8	8
9	9
10	10

- A table with indices $1, 2, \ldots, 10$ is a compact and efficient solution in this case.
- The table above stores the set $\{1, 3, 8, 9, 10\}$.

Tables (3)

Exercise: Describe a dictionary implementation which can store n distinct keys in the range $\{0, 1, 2, 3, \ldots, 2n\}$ with $\Theta(1)$ FIND and INSERT operations.

Tables (4)

Index	**Value**
0	
1	
2	
3	
⋮	⋮
2n-1	
2n	

- Since table lookups are $\Theta(1)$ and $2n \in \Theta(n)$, a table of size $2n$ provides the desired running times and requires $\Theta(n)$ space.
- In general, if a problem requires a dictionary with integer keys in a relatively constrained range, a simple table like the one above is often a good choice.

Tables (5)

Exercise: Describe a dictionary implementation which can store *n* distinct keys in the range $\{0, 1, 2, 3, \ldots, 2^n\}$ with $\Theta(1)$ Find and Insert operations.

Tables (6)

Index	**Value**
0	
1	
2	
3	
⋮	⋮
$2^n - 1$	
2^n	

- In this case, a table would provide the desired running times for Find and Insert, but would require $\Theta(2^n)$ space and initialization time.
- The vast majority of the space would be wasted, since the table would only contain n elements.

Compressed Tables (1)

Index	Value
0	
1	
2	
3	
⋮	⋮
2n-1	
2n	

- **Idea**: Instead of mapping an element *i* to slot *i* of the array, use a different indexing scheme which wastes less space.
- Since most of the table is empty anyway, it should be possible to compress all of the data into a small number of indices.

Compressed Tables (2)

Index	Value
0	10
1	
2	32
3	23
4	
5	
6	6
7	17
8	
9	

- For example, if $n = 5$, the set $\{17, 23, 32, 6, 10\}$, which is in the range $[0, 2^5]$, can be inserted into a table of size 10 by using the last digit as the index.

Compressed Tables (3)

Index	Value
0	10
1	
2	32
3	23
4	
5	
6	6
7	17
8	
9	

- In this example, the index of a particular k is given by the function

$$g(k) = k \bmod 10$$

Compressed Tables (4)

Index	**Value**
0	10
1	
2	32
3	23
4	
5	
6	6
7	17
8	
9	

- Task: FIND(15).
- Since $g(k) = 5$, index 5 is inspected. Since it is empty, the key is not in the dictionary.

Compressed Tables (5)

Index	**Value**
0	10
1	
2	32
3	23
4	
5	
6	6
7	17
8	
9	

- Task: FIND(16).
- Since $g(k) = 6$, index 6 is inspected. Index 6 is not empty, but it does not contain 16, so the key is not in the dictionary.

Compressed Tables (6)

Index	Value
0	10
1	
2	32
3	23
4	
5	
6	6
7	17
8	
9	

- Task: INSERT(16).
- Since $g(k) = 6$, the key 16 should be inserted at index 6.
- But index 6 is full, so insertion is impossible without discarding an existing element.

Compressed Tables (7)

Index	**Value**
0	10
1	
2	32
3	23
4	
5	
6	6
7	17
8	
9	

- **General Issue**: What if every key in the dictionary has the same last digit?
- In practice, it is very common for a data structure to contain a large collection of very similar data.

Compressed Tables (8)

Index	**Value**
0	10
1	
2	32
3	23
4	
5	
6	6
7	17
8	
9	

- The compression approach is useful for reducing table size, but not for spreading similar data out evenly among indices.

Hash Tables (1)

Index	Value
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- **Idea**: What if the indexing scheme incorporates both digits?
- More generally, what if the indexing scheme is designed to incorporate all aspects of each key into its assigned index?

Hash Tables (2)

Index	**Value**
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- **Exercise**: Insert the keys $1, 11, 21, 31$ into the table using the **hash function**

$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

  to determine the index of each key.

Hash Tables (3)

Index	Value
0	
1	1
2	11
3	
4	
5	
6	
7	
8	
9	

$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(1) = (1 + 0) \bmod 10 = 1$
- $h(11) = (11 + 1) \bmod 10 = 2$

Hash Tables (4)

Index	**Value**
0	
1	1
2	11
3	21
4	31
5	
6	
7	
8	
9	

$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(21) = (21 + 2) \bmod 10 = 3$
- $h(31) = (31 + 3) \bmod 10 = 4$

Hash Tables (5)

Index	Value
0	
1	1
2	11
3	21
4	31
5	
6	
7	
8	
9	

$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- Using the function $h(k)$, a set of keys with the same last digit are mapped to different indices of the table.

Hash Tables (6)

Index	Value
0	
1	1
2	11
3	21
4	31
5	
6	
7	
8	
9	

$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- However, there are still multiple keys which receive the same index (such as $2, 11$ and 20).

Hash Tables (7)

A **hash table** is a data structure which implements the dictionary ADT using a table of size M and a hash function $h(k)$ which maps each key k (which may be be of any type) to values in the range $0, 1, \dots, M - 1$.

The set of all possible input keys k is called the **universe** of keys and is denoted by U. The actual set of keys inserted into the table is not necessarily equal to U.

Hash tables are only effective if the structure and size of the input data is reasonably well understood. In particular, the universe U is normally much larger than the table size, so it is necessary to assume that only a small fraction of possible keys will actually be inserted into the table.

Hash Tables (8)

Index	**Value**
0	
1	1
2	11
3	21
4	31
5	
6	
7	
8	
9	

$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- Task: Insert the value 22 into the hash table above.
- $h(22) = (22 + 2) \bmod 10 = 4$.
- The insertion results in a **collision**.

Collisions

A **collision** in a hashing scheme is a pair of keys $k_1, k_2 \in U$ such that

$$h(k_1) = h(k_2)$$

In cases where the table size M is smaller than the number of possible keys in U, collisions are unavoidable due to the Pigeonhole Principle.

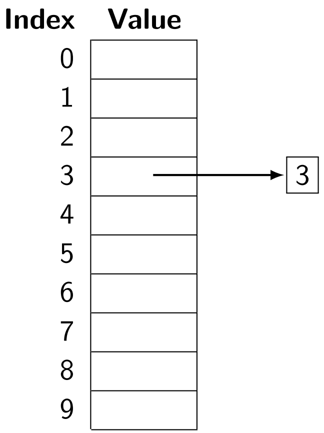
Since collisions are unavoidable, hash tables must include a **collision resolution** scheme to accommodate keys with equal hash values.

Chaining (1)

One simple collision resolution scheme is **separate chaining** (often just called ‘chaining’).

Instead of storing keys in the table itself, keys are stored in linked lists attached to each index of the table. This permits one index of the table to contain multiple keys.

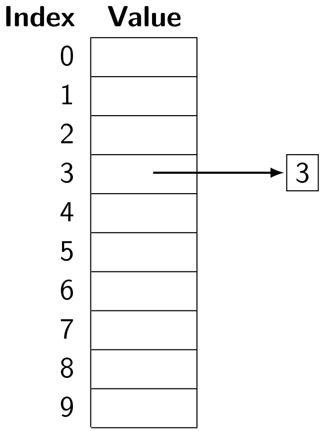
Chaining (2)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- **Example**: Insert the sequence $3, 14, 15, 9, 26, 5, 35$ into the hash table above using chaining.

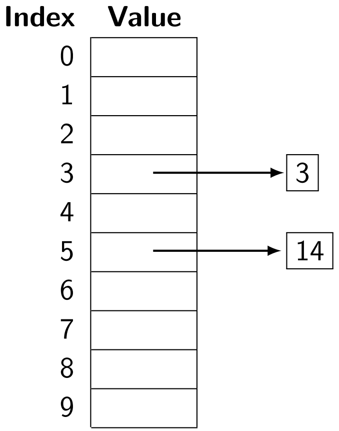
Chaining (3)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(3) = 3$

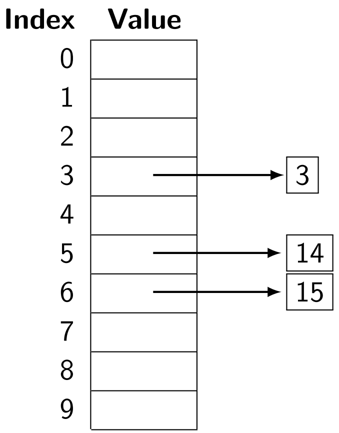
Chaining (4)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(14) = 5$

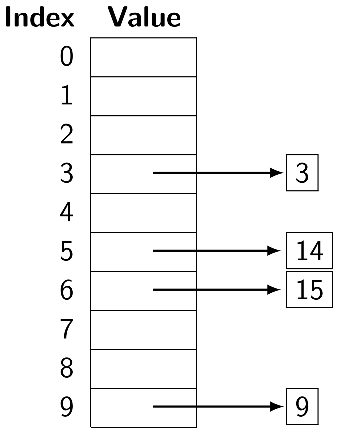
Chaining (5)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(15) = 6$

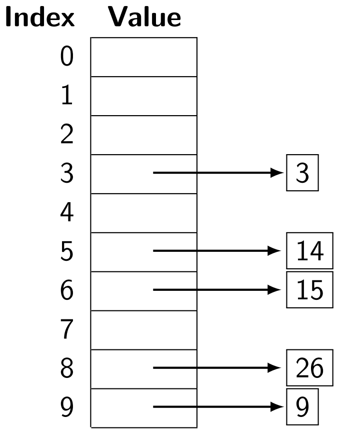
Chaining (6)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(9) = 9$

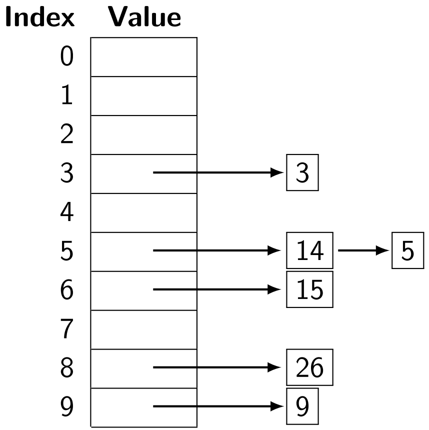
Chaining (7)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(26) = 8$

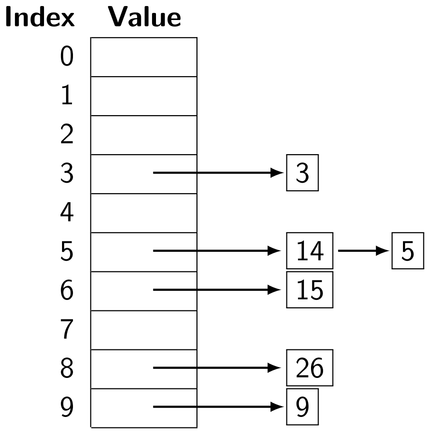
Chaining (8)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(5) = 5$

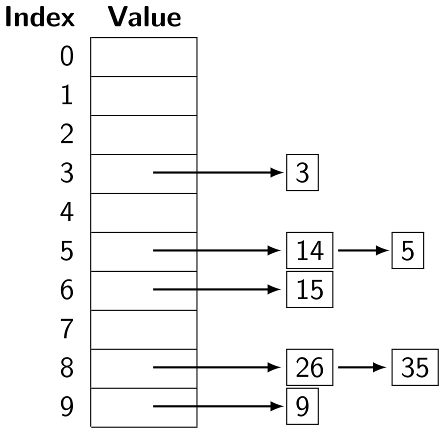
Chaining (9)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- Both evaluating the hash function and insertion into a linked list is $\Theta(1)$, so insertions into a hash table with chaining are $\Theta(1)$.

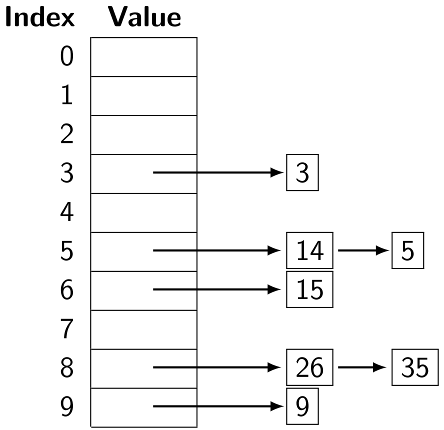
Chaining (10)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(35) = 8$

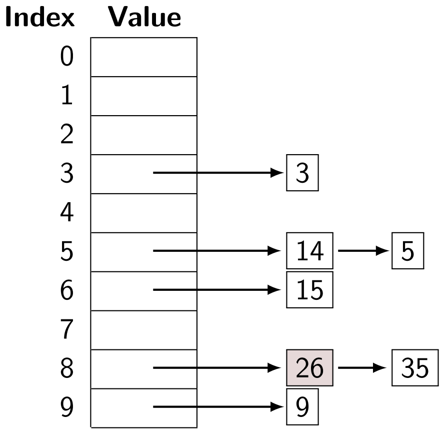
Chaining (11)



$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$

- Task: FIND(35)
- First, evaluate the hash function: $h(35) = 8$.

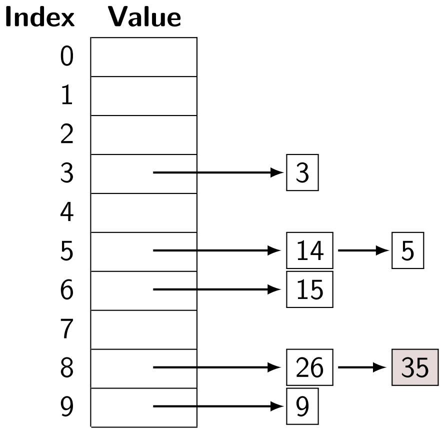
Chaining (12)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- Task: FIND(35)
- Search through the list at index 8.

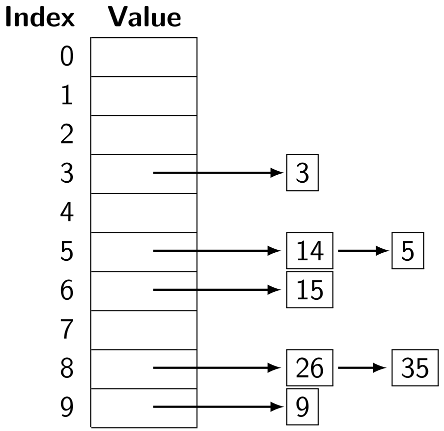
Chaining (13)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- Task: FIND(35)
- If the key is found, return it (and any other associated data).

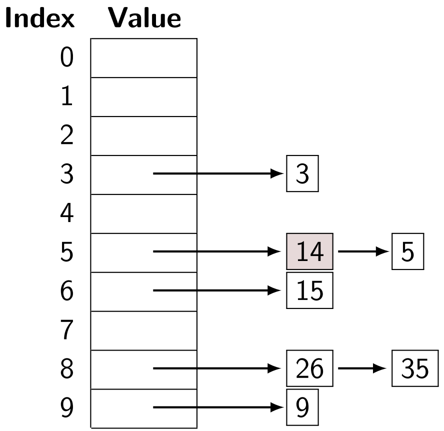
Chaining (14)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- Task: $\textsc{Find}(23)$
- The hash code for 23 is $h(23) = 5$.

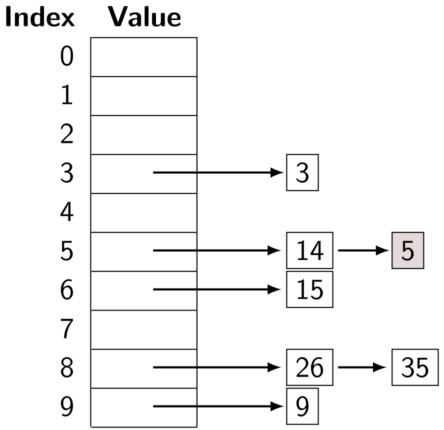
Chaining (15)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- The entire contents of the list at index 5 must be inspected during an unsuccessful query.

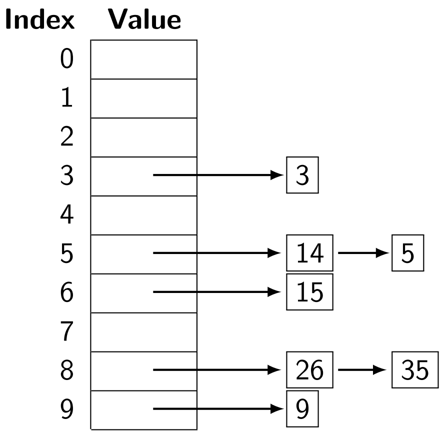
Chaining (16)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- The entire contents of the list at index 5 must be inspected during an unsuccessful query.

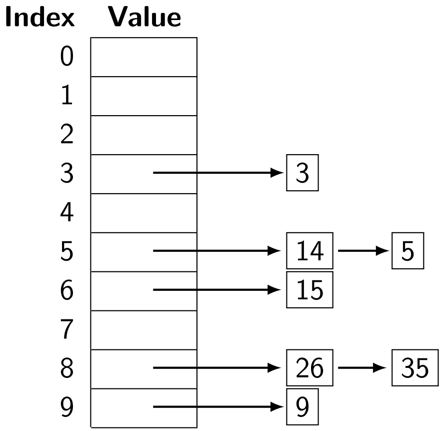
Chaining (17)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- If the key is not found among the items in the list, the key is not in the table.

Chaining (18)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- **Question**: What is the worst-case running time of Find in a hash table with chaining?

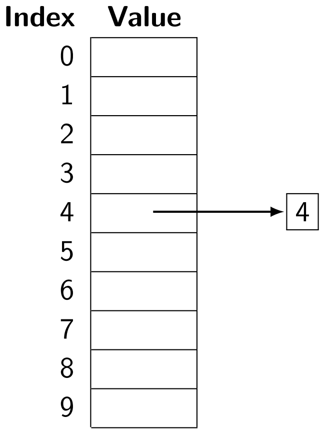
Clustering (1)

Index	Value
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- EXERCISE: Insert the values $4, 13, 22, 31$ into the hash table above, using chaining for collision resolution.

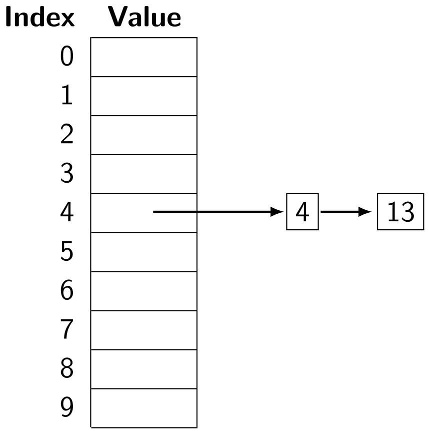
Clustering (2)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(4) = 4$

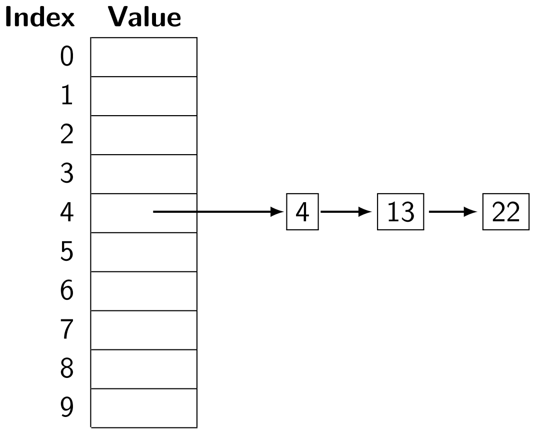
Clustering (3)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(13) = 4$

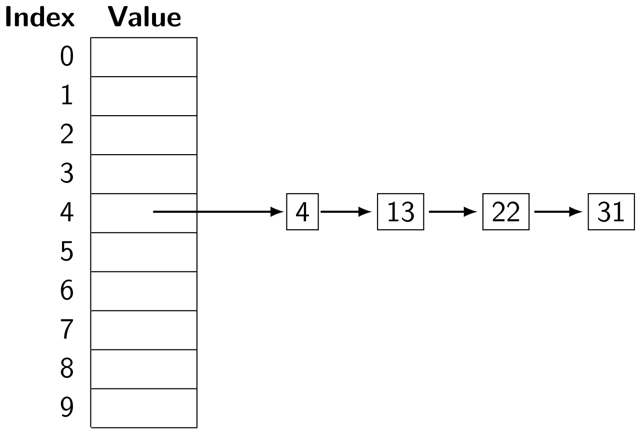
Clustering (4)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(22) = 4$

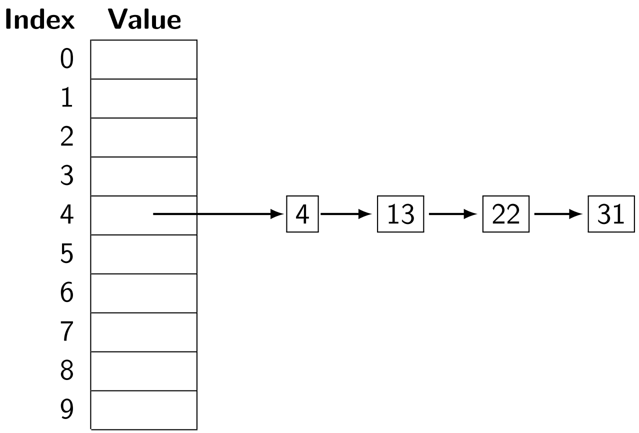
Clustering (5)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- $h(31) = 4$

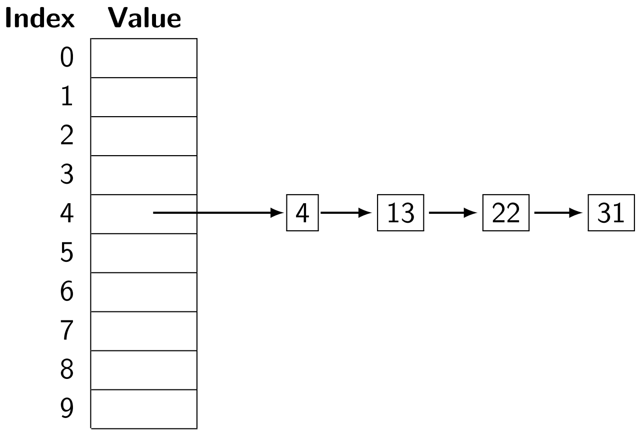
Clustering (6)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- The input data in this example exhibits **clustering** with this hash function.

Clustering (7)



$$h(k) = (k + \lfloor k/10 \rfloor) \bmod 10$$

- Since the data may be clustered, the worst case running time of FIND is $\Theta(n)$.

Clustering (8)

There is no general way to prevent clustering unless all of the input data is known in advance. Collisions are always inevitable, so the goal of a hash function designer must be to minimize the likelihood that similar keys are hashed to similar values.

Ideally, the result of a hash function on a given input sequence should be **indistinguishable from random numbers**. In practice, this is an extremely difficult standard to meet. Hash functions whose output appears to be random are useful for some fields, especially cryptography (CSC 429).

We will focus on choosing hash functions which distribute inputs among indices with equal probability, and which do not have any obvious clustering behavior.

String Hashing (1)

Index	**Value**
0	
1	
2	
3	
4	
5	
6	

- **Exercise**: Design a hashing scheme to insert the strings below into the hash table above (with size 7).

```
ocean  boat   tide   sand   canoe
```

String Hashing (2)

Index	**Value**
0	
1	
2	
3	
4	
5	
6	

- Since the output of a hash function must be an integer index, the first challenge is finding a way to convert compound data types (like strings or structures) into a single hash code.
- The characters of a string are represented by numbers, so we can define a function which uses the numerical values of each character to determine the hash code of the string.

String Hashing (3)

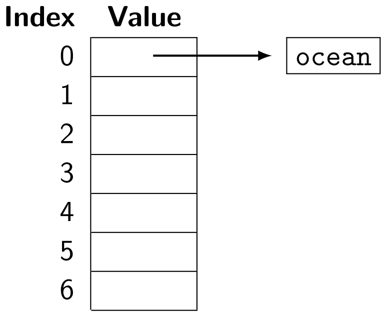
Index	Value
0	
1	
2	
3	
4	
5	
6	

- **First try**: For a string $s = c_1 c_2 c_3 \dots c_k$, define

$$h(s) = (c_1 + c_2 + \dots c_k) \bmod 7$$

- We will see that this is a **bad idea**.

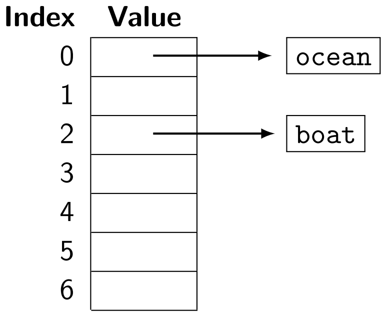
String Hashing (4)



- In the ASCII character set, lowercase letters are numbered starting at 97.

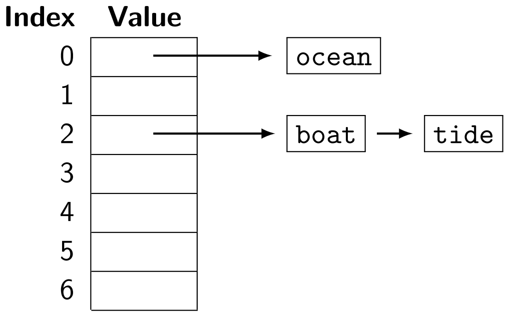
$$\begin{aligned} h(\text{`ocean'}) &= (\text{`o'} + \text{`c'} + \text{`e'} + \text{`a'} + \text{`n'}) \bmod 7 \\ &= (111 + 99 + 101 + 97 + 110) \bmod 7 \\ &= 518 \bmod 7 \\ &= 0 \end{aligned}$$

String Hashing (5)



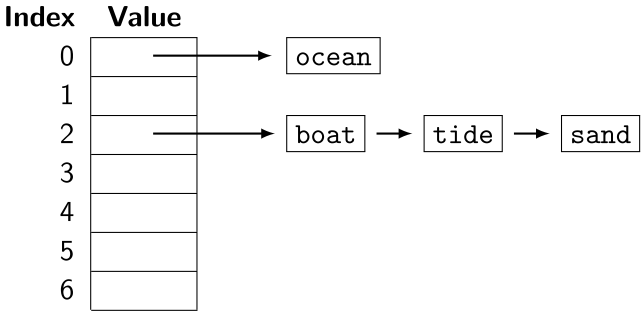
$$\begin{aligned} h(\text{`boat'}) &= (\text{`b'} + \text{`o'} + \text{`a'} + \text{`t'}) \bmod 7 \\ &= (98 + 111 + 97 + 116) \bmod 7 \\ &= 422 \bmod 7 \\ &= 2 \end{aligned}$$

String Hashing (6)



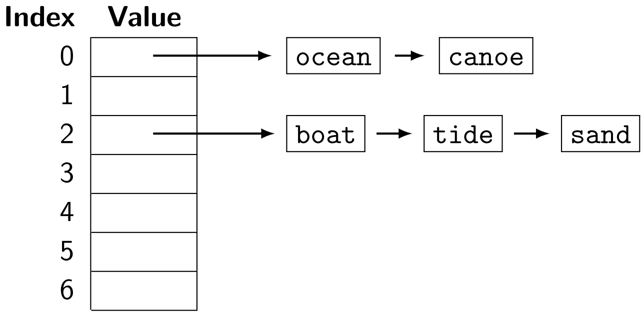
$$\begin{aligned} h(\text{`tide'}) &= (\text{`t'} + \text{`i'} + \text{`d'} + \text{`e'}) \bmod 7 \\ &= (116 + 105 + 100 + 101) \bmod 7 \\ &= 422 \bmod 7 \\ &= 2 \end{aligned}$$

String Hashing (7)



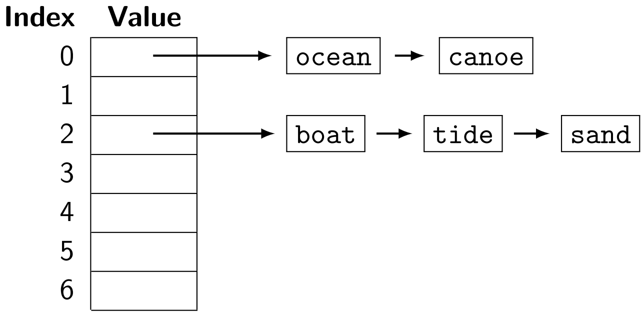
$$\begin{aligned} h(\text{`sand'}) &= (\text{`s'} + \text{`a'} + \text{`n'} + \text{`d'}) \bmod 7 \\ &= (115 + 97 + 110 + 100) \bmod 7 \\ &= 422 \bmod 7 \\ &= 2 \end{aligned}$$

String Hashing (8)



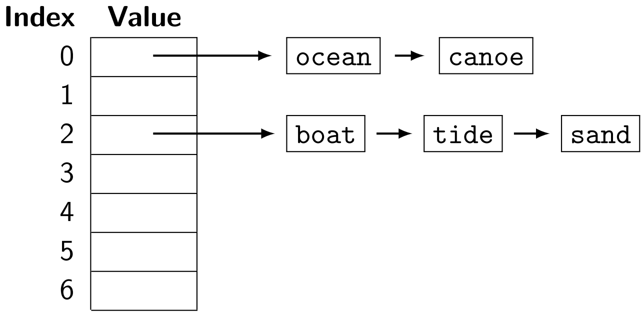
$$
\begin{aligned}
h(\text{`canoe'}) &= (\text{`c'} + \text{`a'} + \text{`n'} + \text{`o'} + \text{`e'}) \bmod 7 \\
&= (99 + 97 + 110 + 111 + 101) \bmod 7 \\
&= 518 \bmod 7 \\
&= 0
\end{aligned}
$$

String Hashing (9)



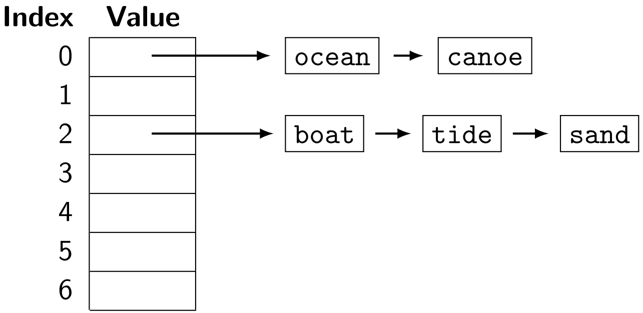
- For any hash function $h(k)$, there will exist input sequences which cause extreme clustering.
- The clustering in index 2 has no obvious cause.
- The two items in index 0 are anagrams, which is evidence of a serious deficiency of the hash function.

String Hashing (10)



$$h(\text{`ocean'}) = (111 + 99 + 101 + 97 + 110) \bmod 7$$
$$h(\text{`canoe'}) = (99 + 97 + 110 + 111 + 101) \bmod 7$$

String Hashing (11)



- Any two words with the same letters will be hashed to the same value, since the hash function does not incorporate any position information into the hash value.

String Hashing (12)

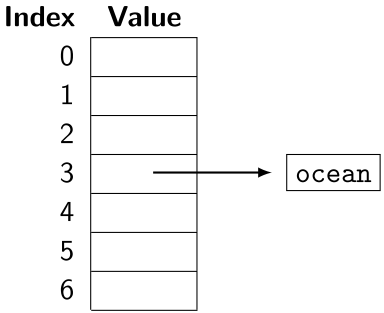
Index	Value
0	
1	
2	
3	
4	
5	
6	

- **Second try**: For a string $s = c_1 c_2 c_3 \ldots c_k$, define

$$h(s) = (c_1^1 + c_2^2 + c_3^3 + \ldots c_k^k) \bmod 7$$

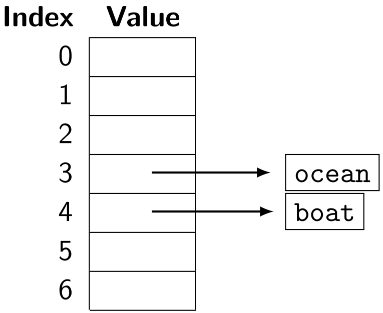
- This hash function weights each letter differently.

String Hashing (13)



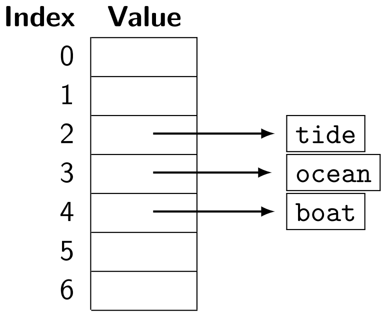
$$\begin{aligned} h(\text{'ocean'}) &= ((\text{'o'})^1 + (\text{'c'})^2 + (\text{'e'})^3 + (\text{'a'})^4 + (\text{'n'})^5) \bmod 7 \\ &= (111^1 + 99^2 + 101^3 + 97^4 + 110^5) \bmod 7 \\ &= 3 \end{aligned}$$

String Hashing (14)



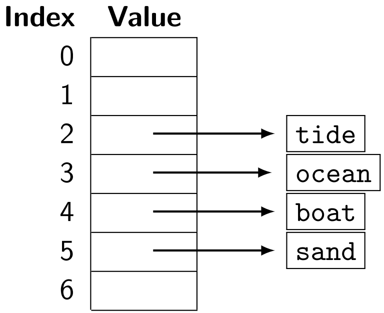
$$\begin{aligned} h(\text{`boat'}) &= ((\text{`b'})^1 + (\text{`o'})^2 + (\text{`a'})^3 + (\text{`t'})^4) \bmod 7 \\ &= (98^1 + 111^2 + 97^3 + 116^4) \bmod 7 \\ &= 181989028 \bmod 7 \\ &= 4 \end{aligned}$$

String Hashing (15)



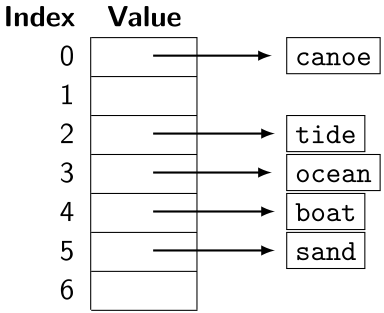
$$\begin{aligned} h(\text{'tide'}) &= ((\text{'t'})^1 + (\text{'i'})^2 + (\text{'d'})^3 + (\text{'e'})^4) \bmod 7 \\ &= (116^1 + 105^2 + 100^3 + 101^4) \bmod 7 \\ &= 105071542 \bmod 7 \\ &= 2 \end{aligned}$$

String Hashing (16)



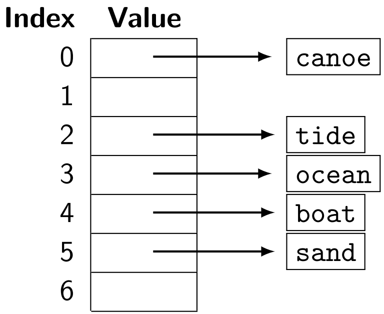
$$\begin{aligned} h(\text{‘sand’}) &= ((\text{‘s’})^1 + (\text{‘a’})^2 + (\text{‘n’})^3 + (\text{‘d’})^4) \bmod 7 \\ &= (115^1 + 97^2 + 110^3 + 100^4) \bmod 7 \\ &= 101340524 \bmod 7 \\ &= 5 \end{aligned}$$

String Hashing (17)



$$\begin{aligned} h(\text{'canoe'}) &= ((\text{'c'})^1 + (\text{'a'})^2 + (\text{'n'})^3 + (\text{'o'})^4 + (\text{'e'})^5) \bmod 7 \\ &= (99^1 + 97^2 + 110^3 + 111^4 + 101^5) \bmod 7 \\ &= 10663248050 \bmod 7 \\ &= 0 \end{aligned}$$

String Hashing (18)



- The second hash function produces an optimal distribution.
- With a well-chosen hash function, hash tables have $\Theta(1)$ expected running times for both Insert and Find.
- Experimental analysis is the often best way to find the best hash function for a particular task.

Load Factor

Some degree of clustering is inevitable with any hashing scheme, especially when the table size is not much larger than the number of entries.

The **load factor** of a hash table of size M containing n keys is

$$\alpha = \frac{n}{M}.$$

Rule of thumb: Choose the table size to keep $\alpha \leq 0.6$.