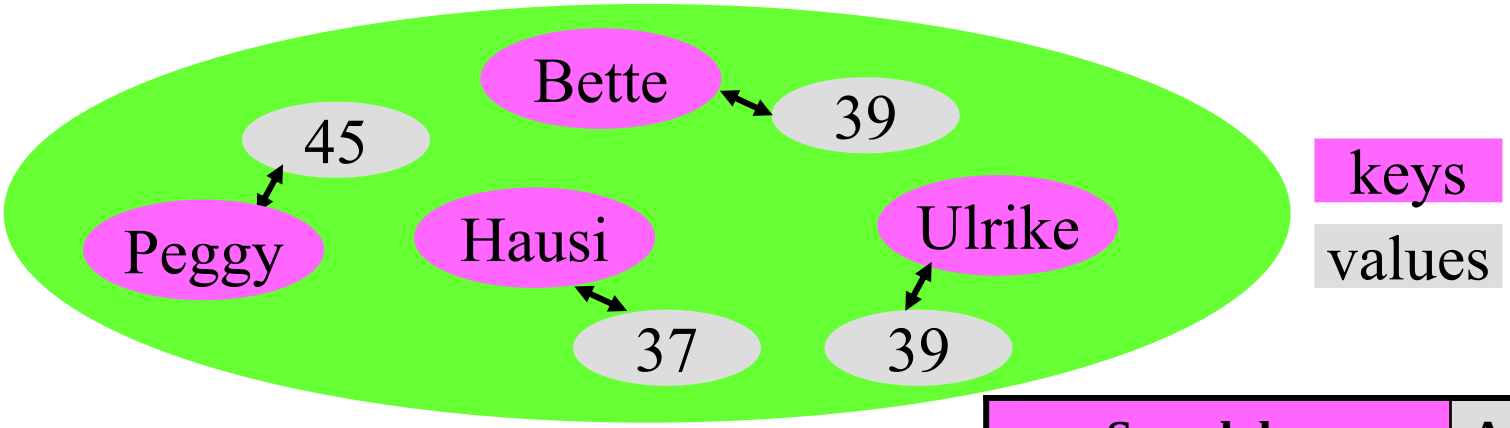
CSC 226

Algorithms and Data Structures: II

Rich Little
rlittle@uvic.ca

Dictionary (Map)

- A ***map*** or ***dictionary*** is a container that contains key-value pairs, (k, v)
- The keys can be unique, but the values can be anything (e.g., don't have to be unique)



Search key	Associated value
Bette	39
Hausi	37
Peggy	45
Ulrike	39

Map Data Structure

- A map data structure, M, supports fundamental methods:
  - `get(`k`)`: If M contains item with key k return v, else NULL.
  - `put(`k`,`v`)`: Insert item with key k and value v; if exists, replace current value with v.
  - `remove(`k`)`: Find (k,v) in M, remove and return it. If (k,v) is not in M, return NULL.

Implementation Strategies

- Ordered Dictionaries:
  - *Lookup Table* - We store the items in sorted order by key linearly (i.e. as a sequence)
  - *Binary Search Tree* - a binary tree storing keys (or key-element pairs) at its internal nodes
- Unordered Dictionaries:
  - *Log File* - We store the items in a sequence in arbitrary order
  - *Hash Table* – elements are stored in an addressable table based on keys.

Log File

- A log file is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked lists or a circular array), in arbitrary order
- Performance:
  - `put`(k, v) takes $\boldsymbol{O(1)}$ time since we can insert the new item at the beginning or at the end of the sequence
  - `get`(k) and `remove`(k) take $\boldsymbol{O(n)}$ 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 log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Lookup Tables

- Suppose that the universe of keys is the set of integers in the range $[0, N-1]$.
- We can allocate an array of size N, where each cell holds a *key-value* pair.
- We do so by placing (k, v) in cell $A[k]$.
  - This is a ***lookup table***
- Each of the fundamental methods runs in $O(1)$ time in the worst-case.
- However, the total memory used in this implementation is $\theta(N)$
  - Is this a problem?

Lookup Tables

- **Exercise:** Describe a dictionary implementation which can store 5 distinct keys in the range [0,10].

Lookup Tables

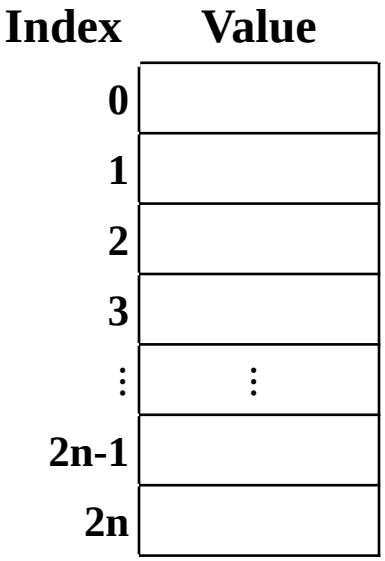
Index	Value
0	
1	1
2	
3	3
4	
5	
6	
7	
8	8
9	9
10	10

- A table with indices 0,1,2,...,10 is a compact and efficient solution in this case.
- This table stores the set {1,3,8,9,10}

Lookup Tables

- **Exercise:** Describe a dictionary implementation which can store n distinct keys in the range $[0,2n]$ with $O(1)$ operations.

Lookup Tables

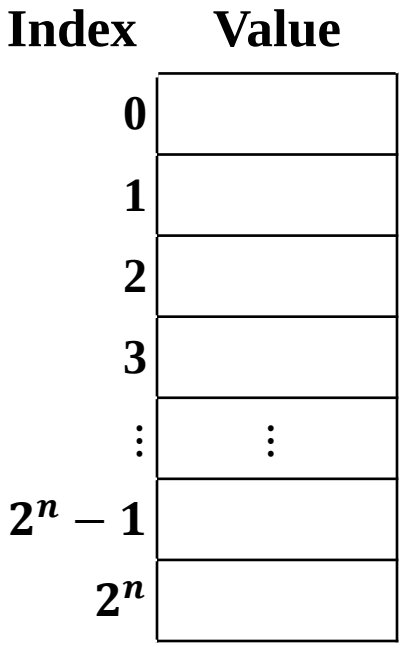


- Since table lookups are $O(1)$ and $2n \in \Theta(n)$, a table of size $2n$ provides the desired running times and requires $\Theta(n)$ space.
- In general, if a problem requires a dictionary with integer keys in a relatively constrained range, a simple table like the one above is often a good choice.

Lookup Tables

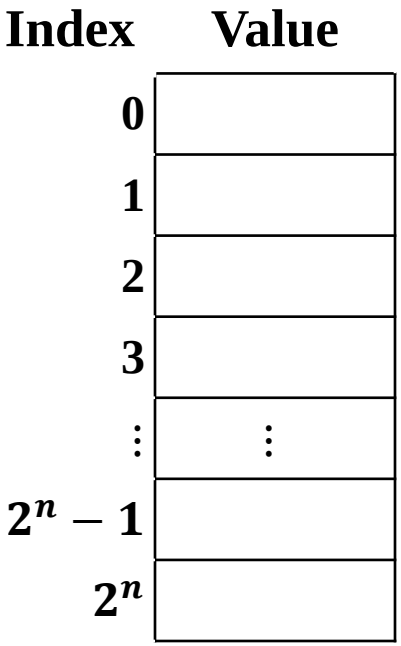
- **Exercise:** Describe a dictionary implementation which can store n distinct keys in the range $[0, 2^n]$ with $O(1)$ operations.

Lookup Tables



- In this case, a table would provide the desired running times, 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.

Lookup Tables



- **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

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

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

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

Compressed Tables

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

- **Task:** `get(15)`
- Since $h(15) = 5$, index 5 is inspected. Since it is empty, the key is not in the dictionary.

Compressed Tables

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

- **Task:** `get(16)`
- Since $h(16) = 6$, 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

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

- **Task:** `put(16)`
- Since $h(16) = 6$, key 16 should be inserted at index 6. But index 6 is full, so insertion is impossible without discarding an existing element.

Compressed Tables

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

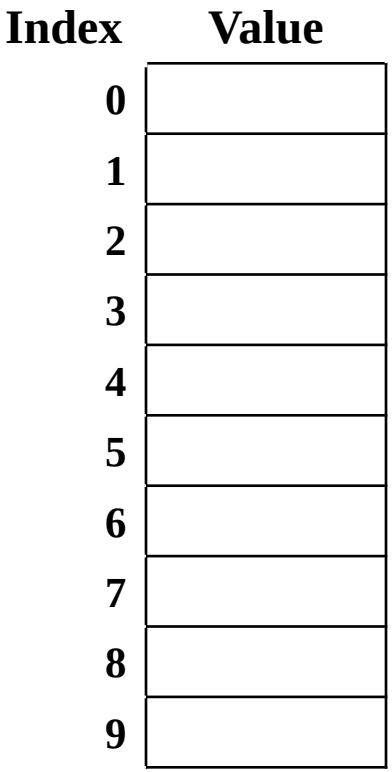
- **General Issue:** What if every key in the dictionary has the same last digit?
- In practice, it is very common for a data structure to contain a large collection of very similar data.

Compressed Tables

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



- **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

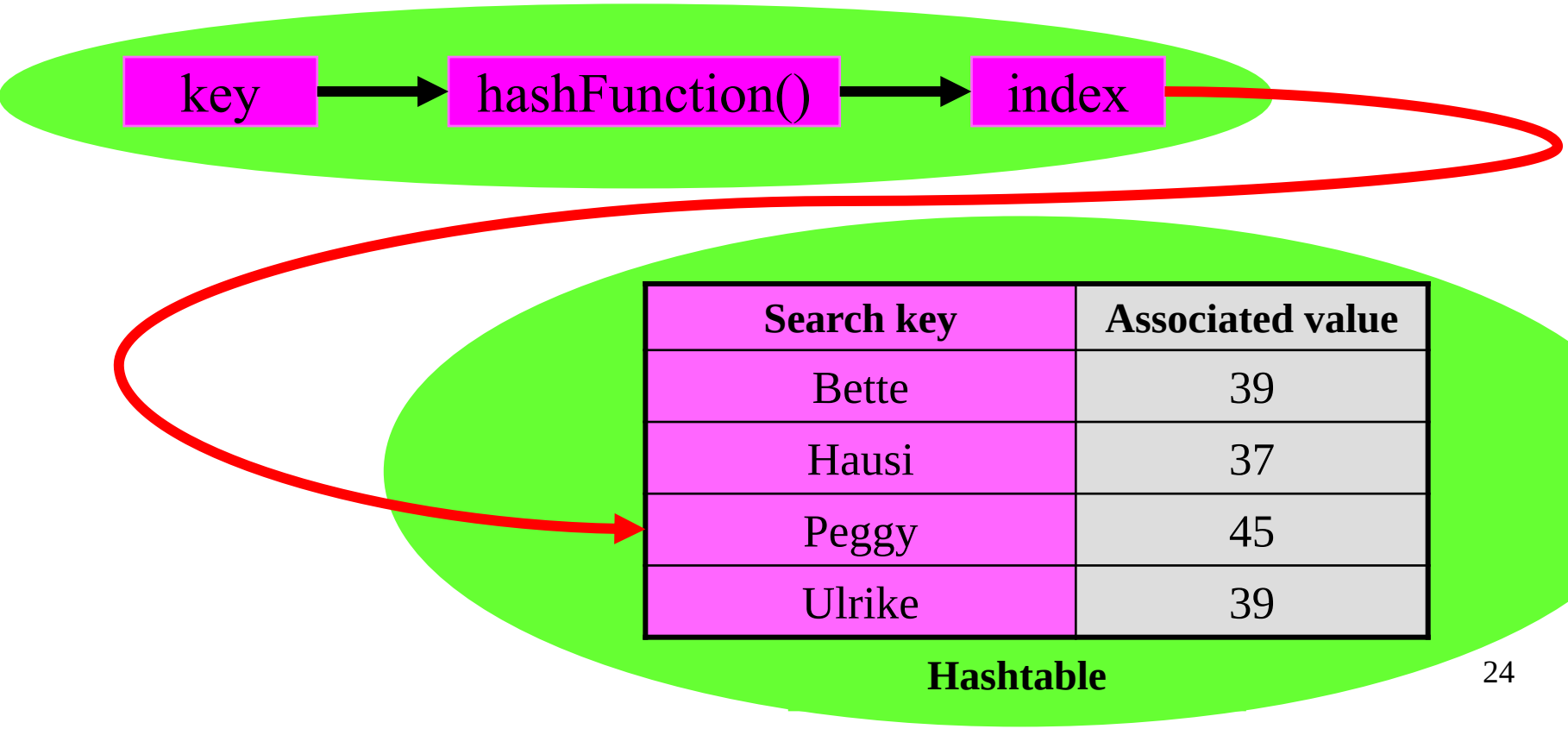
- A ***hash table*** is a data structure which implements the dictionary ADT using a table of size N and a hash function $h(k)$ which maps each key k (which may be of any type) to values in the range $0,1,\ldots,N-1$.

Hash Tables

- 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.

How does hashing work?

- How can we find an element in a hash table in time $O(1)$?
- Given a key, we compute an index into the hash table using a ***hash function***



Hash Function

- A ***hash function*** maps keys to indices
  - It should map keys uniformly across all possible indices
  - It should be fast to compute
  - It should be applicable to all objects

- A hash function is usually specified as the composition of two functions:
  - Hash code map:
    $\boldsymbol{h}_1$: keys $\rightarrow$ integers
  - Compression map:
    $\boldsymbol{h}_2$: integers $\rightarrow [0, \boldsymbol{N} - 1]$

Hash Tables

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**

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

Hash Tables

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

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).
- These are called ***collisions***.

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 N 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.

Collision Resolution

- Two approaches to resolve collisions
  - Separate chaining
    - Store all elements which map to the same location in a linked list
  - Open addressing
    - Check whether other cells of the hash table are free in a given order

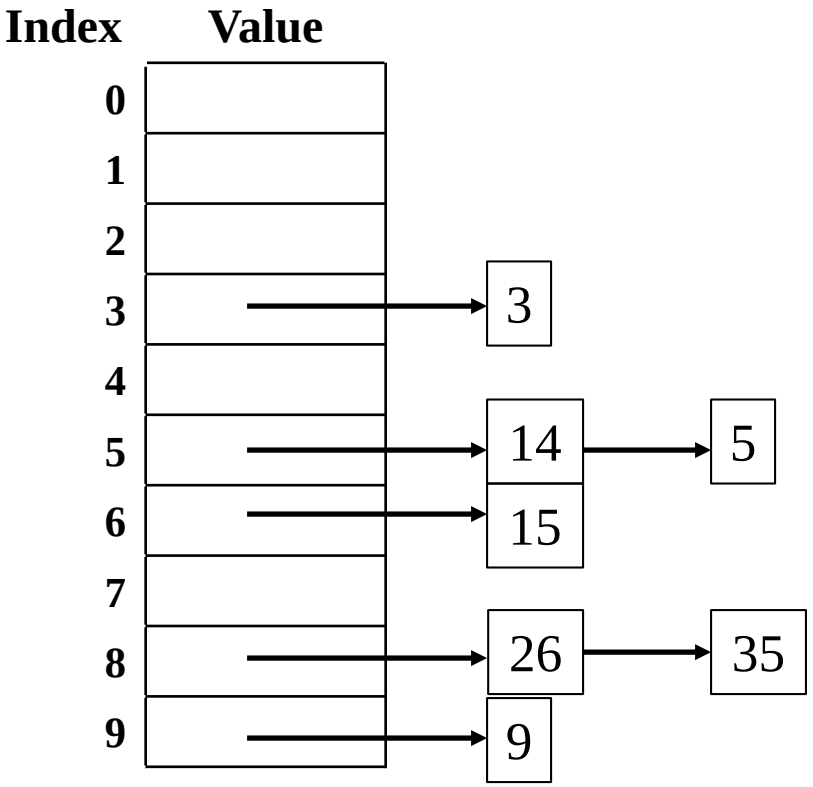
Separate Chaining

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

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

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

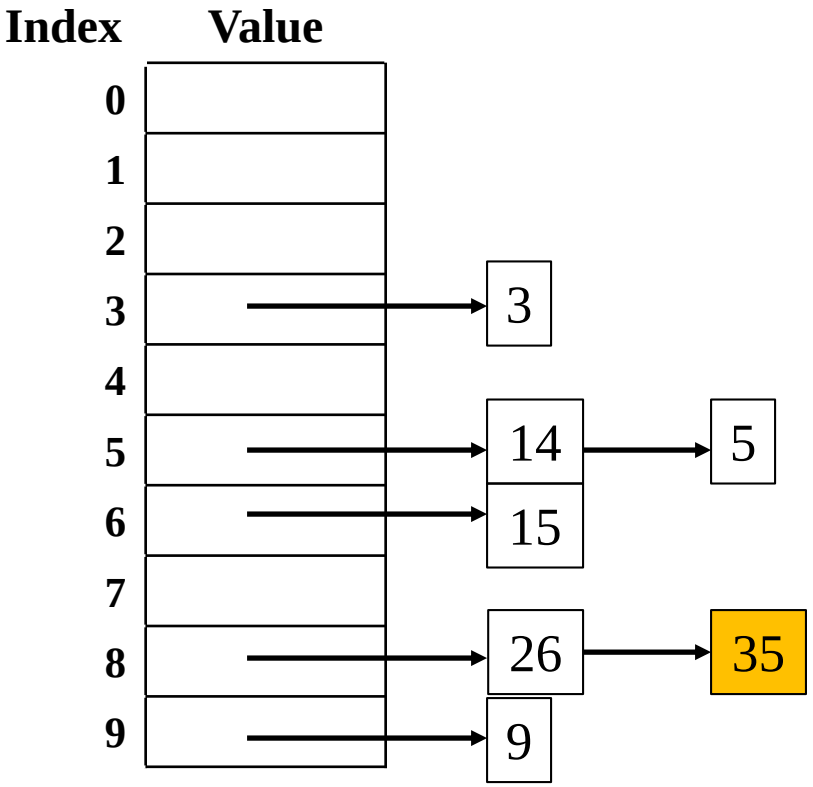
Separate Chaining



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

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

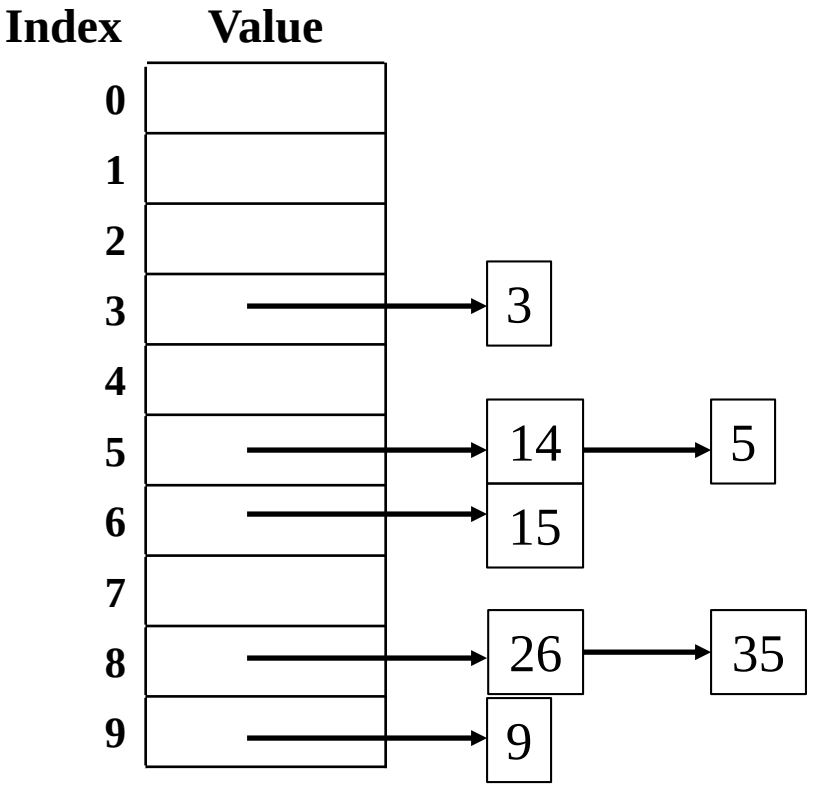
Separate Chaining



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

- Task: `get(35)`
- First, evaluate the hash function: $h(35) = 8$.
- Search through the list at index 8.

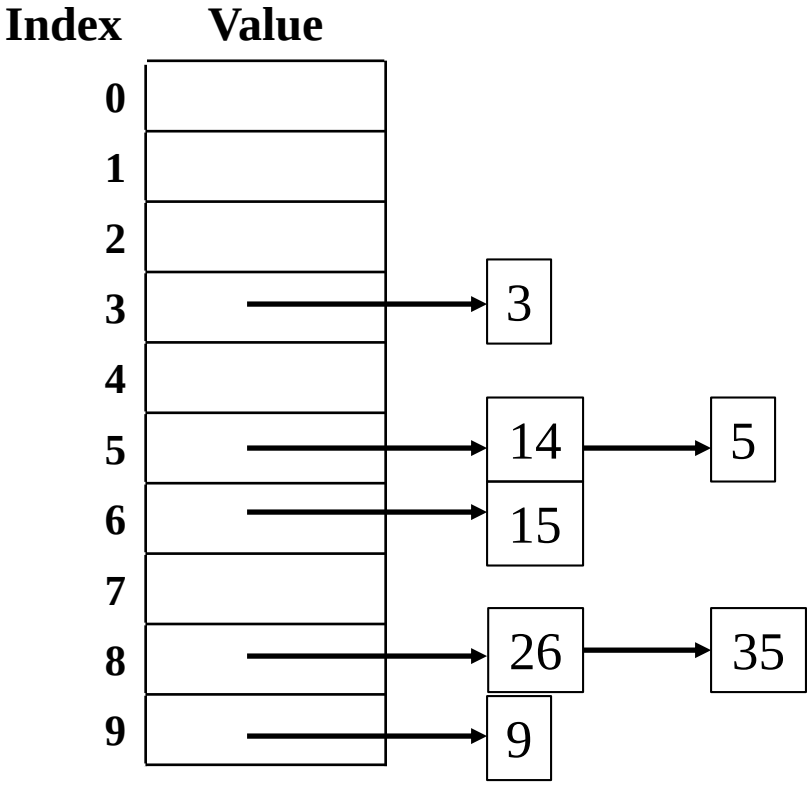
Separate Chaining



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

- Task: `get(23)`
- hash function: $h(23) = 5$.
- Search through the list at index 5…not there, thus not in table

Separate Chaining



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

- **Question:** What is the worst-case running time of `get()` in a hash table with chaining?

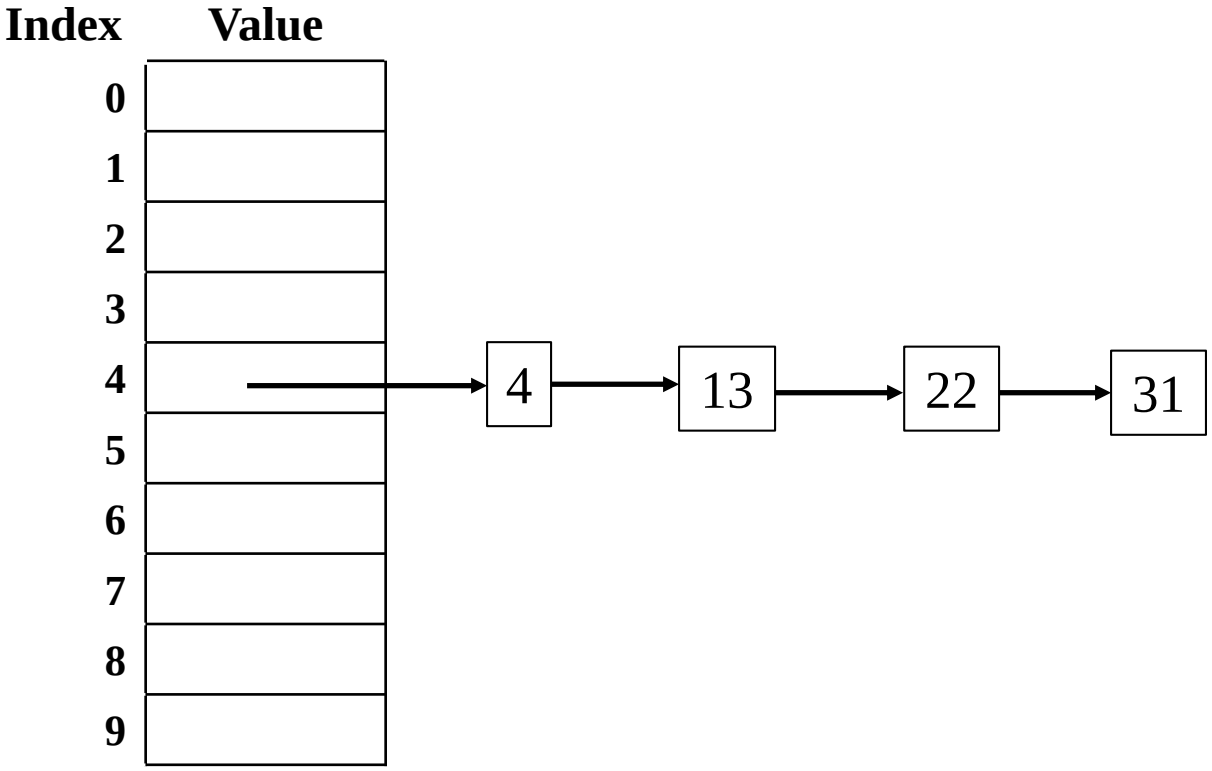
Separate Chaining

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.

Separate Chaining



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

- The input data in this example exhibits ***clustering*** with this hash function.
- The worst case running time of `get()` is $O(n)$.

Clustering

- 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).

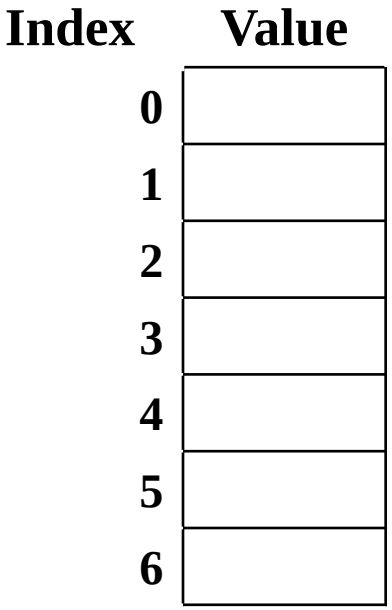
String Hashing

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



- 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.

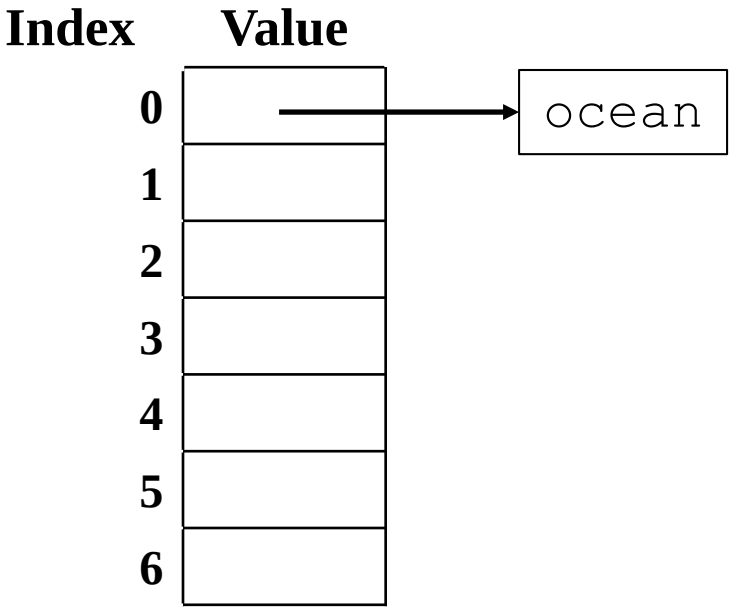
String Hashing

Index	Value
0	
1	
2	
3	
4	
5	
6	

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

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

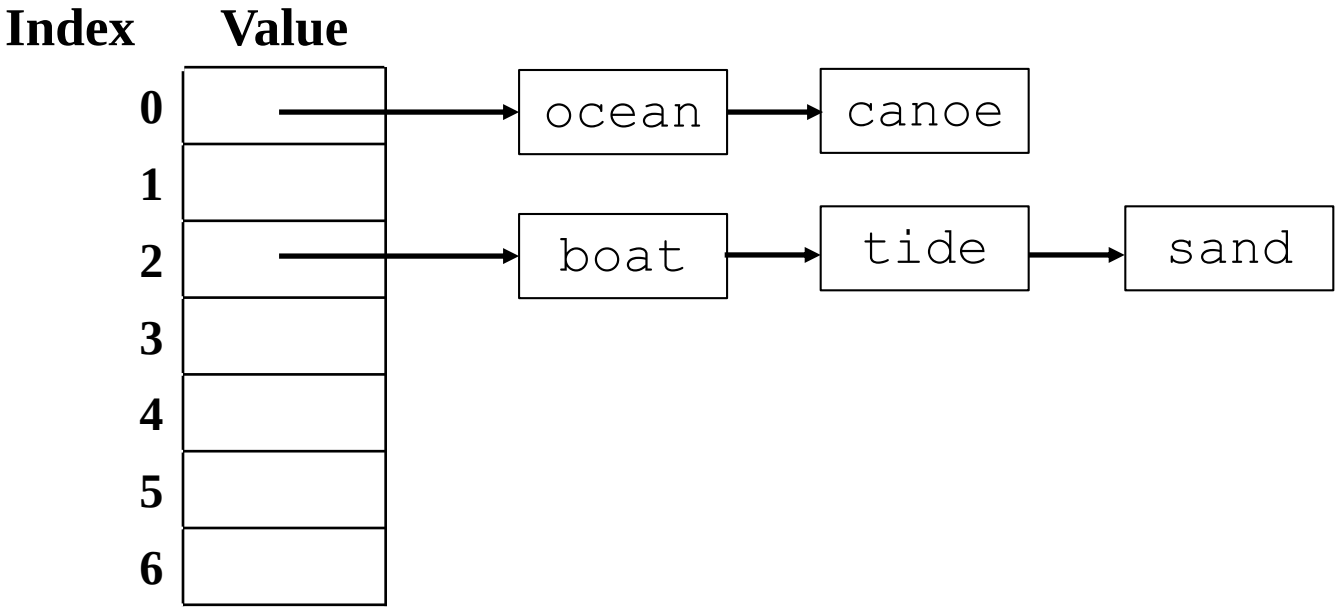
String Hashing



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

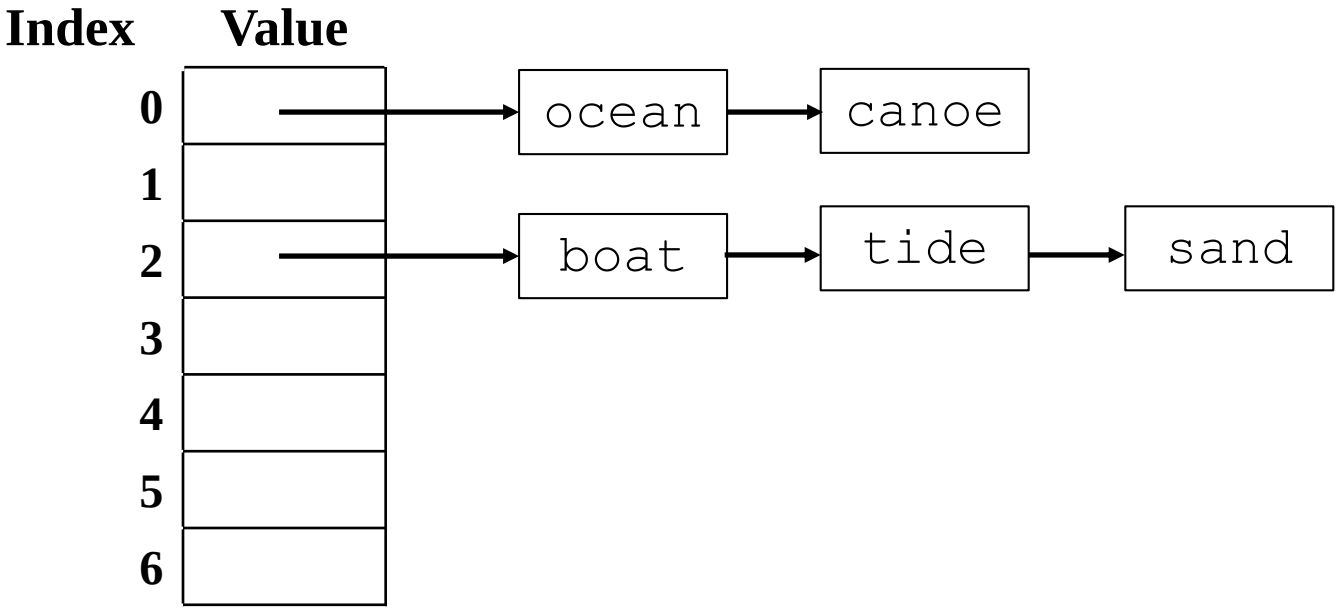
$$\begin{aligned} h('\text{ocean}') &= ('o' +' c' +' e' +' a' +' n') \bmod 7 \\ &= (111 + 99 + 101 + 97 + 110) \bmod 7 \\ &= 518 \bmod 7 \\ &= 0 \end{aligned}$$

String Hashing



- 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



- 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

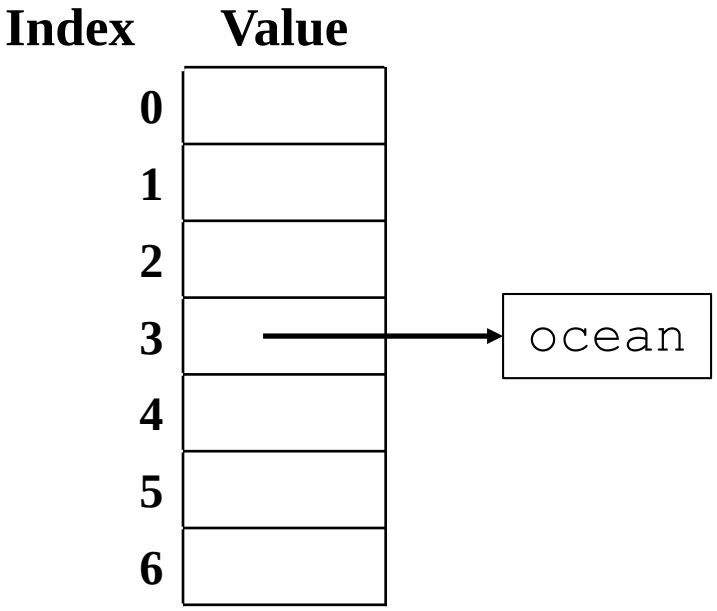
Index	Value
0	
1	
2	
3	
4	
5	
6	

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

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

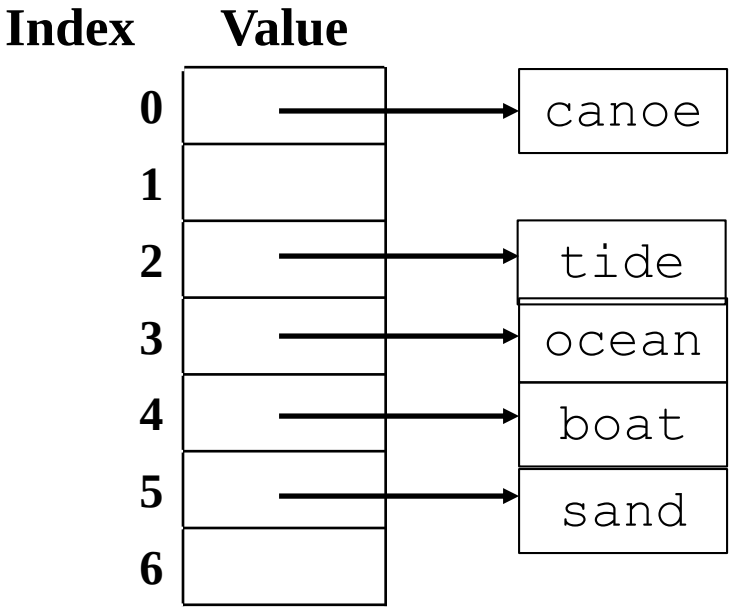
- This hash function weights each letter differently

String Hashing



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

String Hashing



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

Java Conventions

- In Java every data type inherits a method called `hashCode()`
  - returns a 32-bit integer
- this is a hash code NOT a full hash function
- You need to combine `hashCode()` with modular hashing to create an index.
- For example,

```
private int hash(Key x) {
   return(x.hashCode() & 0x7fffffff) % M
}
```

Example: Strings in Java

- `java.lang.String` implements `hashCode()` on a string s of length n by `int` arithmetic using,

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$
$$= s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \cdots + s[n-1]$$

- Of course, you can override this method with your own hash code