Python for Data Scientists L12 : Hash function and hash table

Shirin Tavara

Some of the slides are by Sedgewick & Wayne

And Paul Kube, University of California, San Diago

Outline of the lecture

Direct addressing

Hashing

- Hash function, and hash table, collisions, chaining, and linear probing

Applications

Direct Addressing - Array

Direct addressing. Allocate an array that has one position for every possible key.

Suppose we want to maintain a list of 250 IP addresses (IP: 32 bits)

$$\text{IP: } 128.24.168.01$$

For the fast lookup

- Create an array indexed by IP address
- Size of the array?
- $2^{32} \approx 4 \times 10^{9}$ possible IP addresses -> entries in the array
- Majority of the entries will be blank

What to do?

Linked List

Another approach: use a linked list.

Size?

- A linked list of 250 records

Problem?

- Very slow accessing records, time will be proportional to 250 customers

How to get the best of both array and linked list?

- Amount of memory proportional to #customers!
- Fast access and lookup time!

2^{32} possible IP addresses and 250 customers

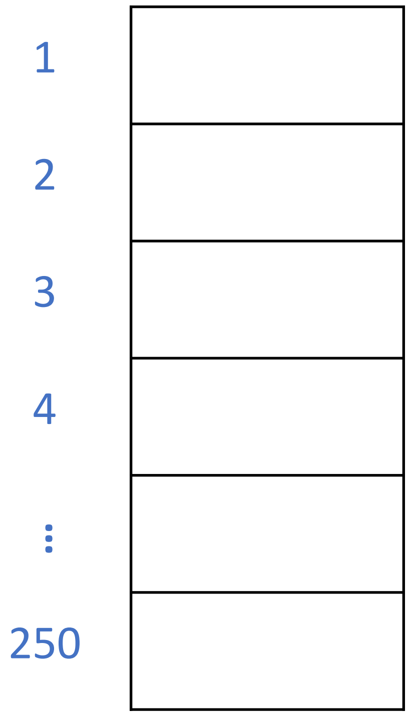
Memory proportional to #customers

- Give a number between 1 and 250 to each of the 2^{32} possible IPs

Issue?

- Many IPs will have the same number
- Hopefully, most of the IP addresses of our customers will be distinct numbers

Store the records in an array of size 250 indexed by the numbers between 1 and 250



Finding data fast

Search for a key in

- An unsorted array or a linked list is *O*(N) on average
- A sorted array or a balanced/randomized search tree is *O*(log N) on average

Can a search be better?

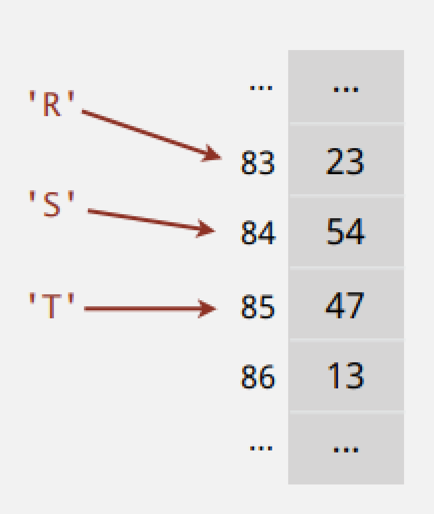
- Yes, if we know key *k* is in the array at index *i*, then *O*(1). But it is unreasonable just know the index.
- What if we can compute the index in the array?
- Hashing and hash tables are ways to implement this idea in which it is possible to do the search, find and delete operations in *O*(1)

Example: Calculating Character Frequencies

Problem. Calculate statistics over the characters that occur in a text.

- If the text only contains ASCII values (the first 128 Unicode characters), then we can use an array of size 128 and store the frequencies.

```
public int[] calcFrequencies(String text) {
    int[] freqs = new int[128];
    for (int i = 0; i < text.length(); i++) {
        int key = (int) text.charAt(i);
        freq[key]++;
    }
    return freqs;
}
```



- But what if the text can contain any Unicode character? (There are 1 114 112 of them…)

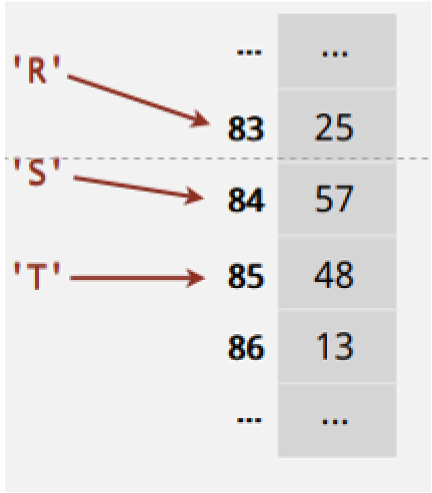
- If we try to use an array with size 1 114 112, it will use a lot of memory, and it will be very sparse (many blank cells in the array).

Example: Calculating Character Frequencies

Problem. Calculate statistics over the characters that occur in a text.

- If the text only contains any Unicode value, then we can still use an array of size 128 and store the frequencies
- We can use (key % 128) to get the array index

```
public int[] calcFrequencies(String text) {
    int[] freqs = new int[128];
    for (int i = 0; i < text.length(); i++) {
        int key = (int) text.charAt(i);
        key = key % 128;
        freq[key]++;
    }
    return freqs;
}
```

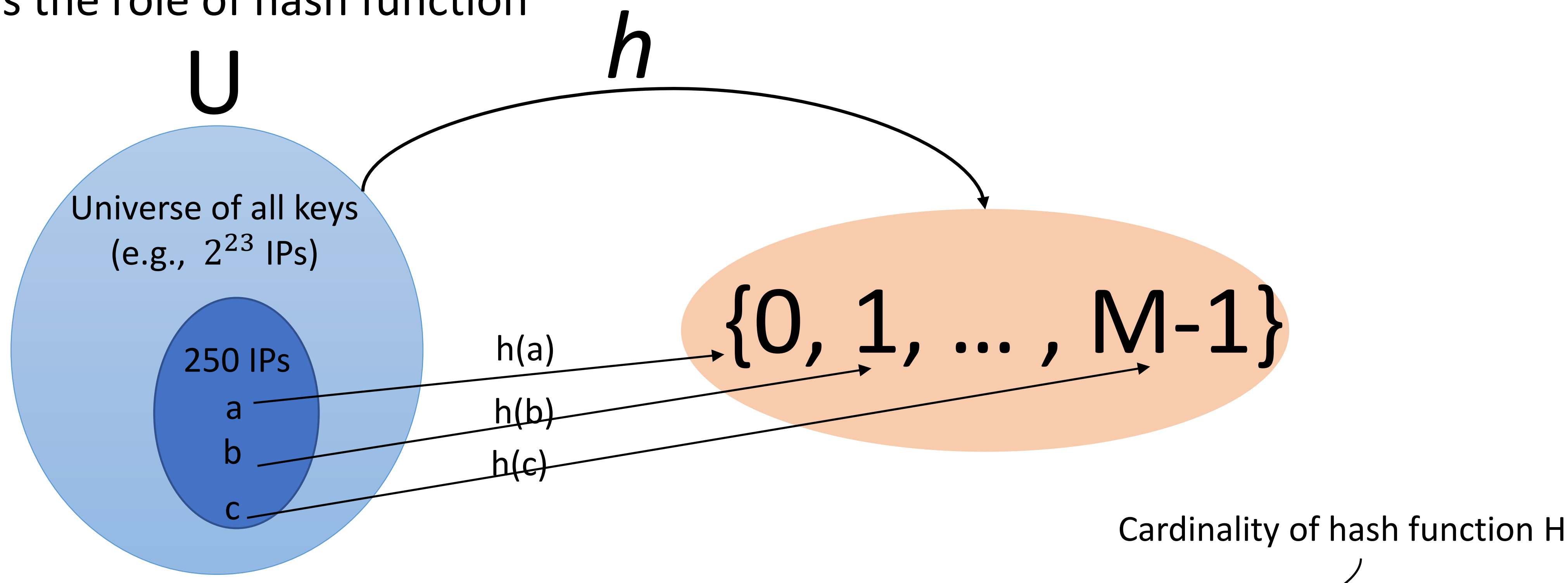


Hash Function

How to assign a number to each key?

- e.g., an IP address or a character?

Answer: it is the role of hash function

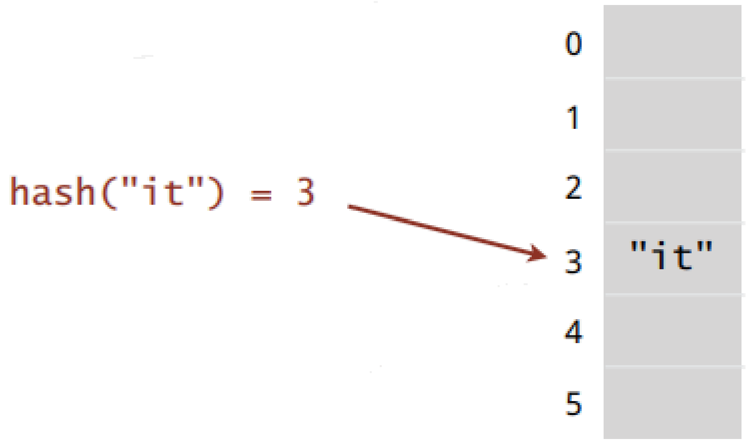


$h: U \rightarrow \{0, 1, \ldots, M-1\}, M>0$

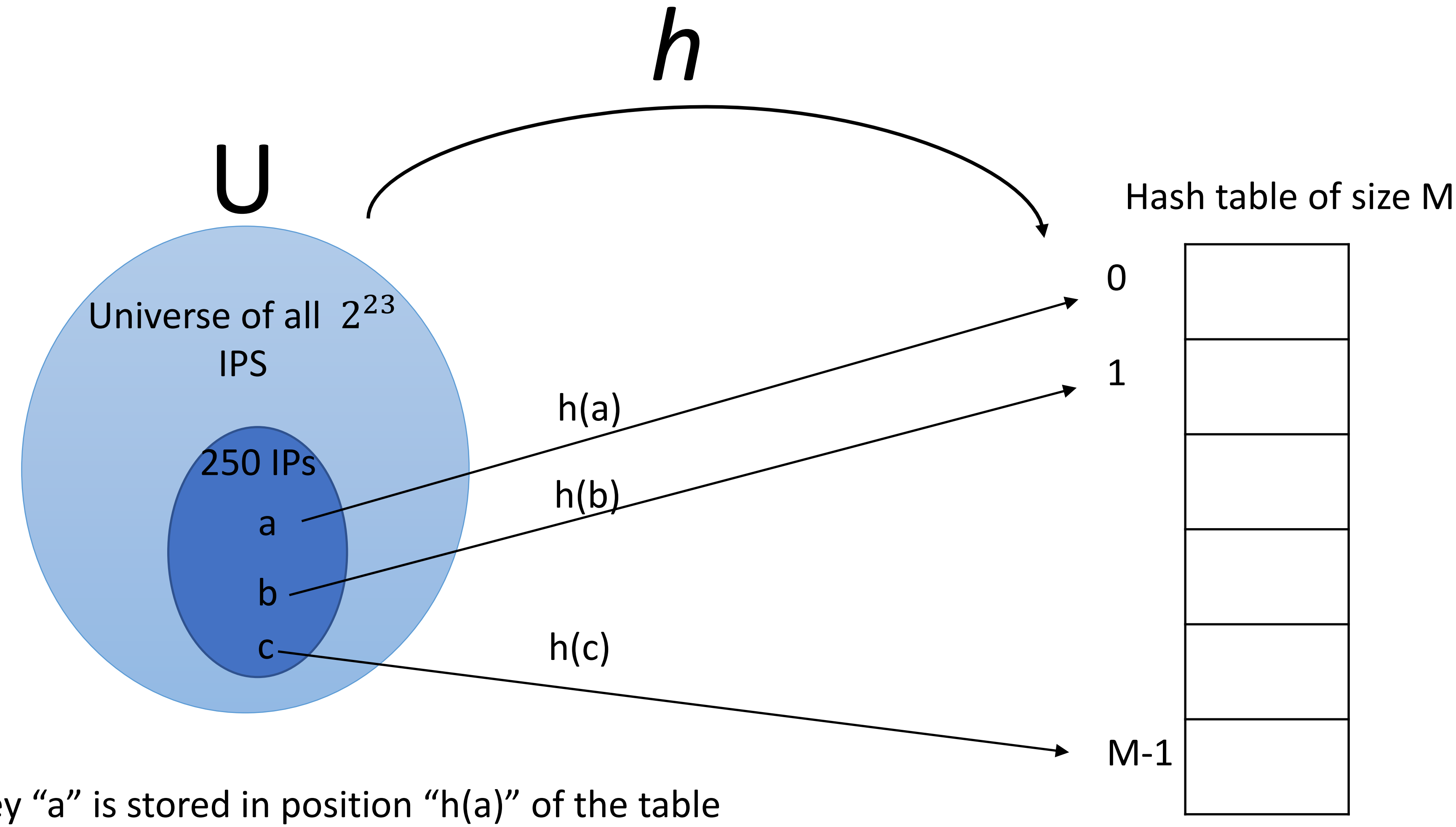
Hashing

Hash function. A method for computing the array index from the key.

Hash table. Stores key or key/value pairs in a key-indexed table in which the address or the index value of the data element is generated from a hash function (index is a function of the key).



Hash Table



The value for key “a” is stored in position “h(a)” of the table

Hash Tables - Example

(tomato, red)

hash of ‘tomato’ -> 26

26 mod 5 = 1

(spinach, green)

hash of ‘spinach’ -> 30

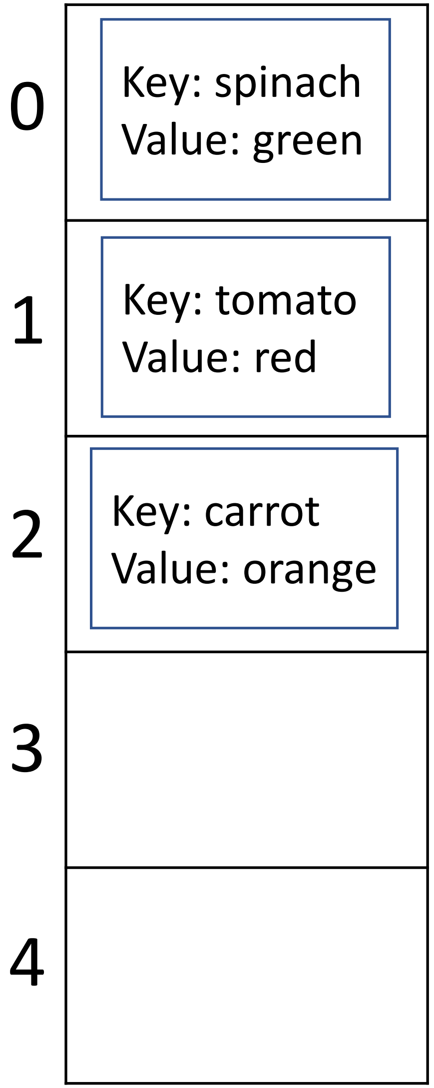
30 mod 5 = 0

(carrot, orange)

hash of ‘carrot’ -> 22

22 mod 5 = 2

Hash Table

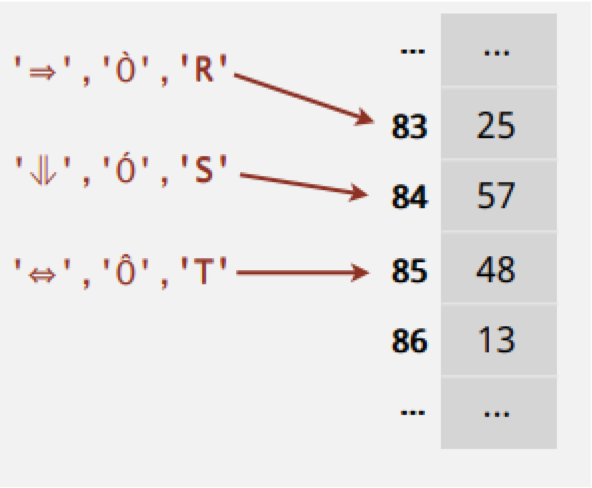


Example: Calculating Character Frequencies

Problem. Calculate statistics over the characters that occur in a text.

- If the text only contains any Unicode value, then we can still use an array of size 128 and store the frequencies
- We can use (key % 128) to get the array index

```
public int[] calcFrequencies(String text) {
    int[] freqs = new int[128];
    for (int i = 0; i < text.length(); i++) {
        int key = (int) text.charAt(i);
        key = key % 128;
        freq[key]++;
    }
    return freqs;
}
```



- But there will be conflicts!
  e.g., ⇒, Ò and R will all map to the same index in the table, i.e., 83

Question. How do we handle conflicts?
Question. Can we use the same idea for any kind of objects?

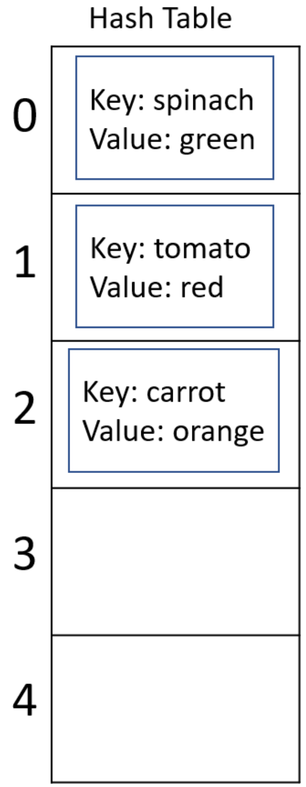
Hash Functions

Load factor:

$$load\ factor = \frac{number\ of\ (key, value)\ pairs}{number\ of\ cells\ in\ the\ hash\ table\ (buckets)} = \frac{N}{M}$$

Example: $load\ factor = \frac{3}{5} = 0.6$

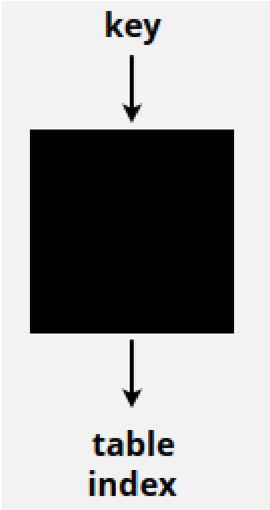
- The load factor is a measure of how full the table is



Hash Function – Desired Properties

Idealistic goal.

- Keys uniformly to produce a table index.
- h: fast and efficiently computable.
- Each table index should be equally likely for each key.
- Direct addressing with *O*(M) memory
- Small M is desirable
- |U| <= M



Question Poll

Which of the following sentence is correct?

1) Hash function should be totally random.
2) Hash function should not be random.
3) Hash function should be in some sense and a consistent function

Question Poll

Which of the following sentence is correct?

1) Hash function should be totally random.
2) Hash function should not be random.
3) Hash function should be in some sense and a consistent function

Hash Function should in some sense be random, and consistent that we can get the same value each time

Question – Poll

Which of the following hash function is a better function compared to others?

1) h(031-422-24)= 031 (the first 3 digits represent the city code)
2) h(031-422-24)= 24
3) h(Personal Number) = first 4 digits, i.e., 1990-0603-xxxx
4) h(Personal Number) = last 4 digits, i.e., 1990-0603-xxxx

Designing Hash Functions

Ex1: IP address.

- Bad: h(128.24.168.70) =128 ← most customers from Asia
- Better: h(128.24.168.70) =70

Ex2: Phone numbers.

- Bad: first three digits. ← most Swedish phone number starts with the same first 3 digits
- Better: last three digits.

Ex3: Personal number.

- Bad: first three digits. ← Personal number always starts with 19xx or 20xx, birth year
- Better: last three digits.

Hash functions for integers

- A general hash function for integer keys and a table of size M (a prime number) is h(k) = k mod M

- If table size is not a prime, this may not work well for some key distributions
  - If table size is an even number; and keys happen to be all even, or all odd. Then only half the table locations will be hit by this hash function.

- So, use prime-sized tables with h(k) = k mod M

Random Hash Functions

- A hash function tries to distribute keys "randomly" over table locations
- For typical integer keys k, with prime table size M, h(k) = k mod M, usually does a good job.
- But with any hash function, it is possible to have "bad" behavior, where most all keys the user wants to insert in the hash table, hash to the same location
- To fix this, the hash function can be a pseudorandom number generator whose output depends on a "random" seed decided when the table is created, and on the key value
  - this is called "random hashing“
  - Random hashing is mostly of theoretical interest; in practice, other simpler hash functions give results that are as good

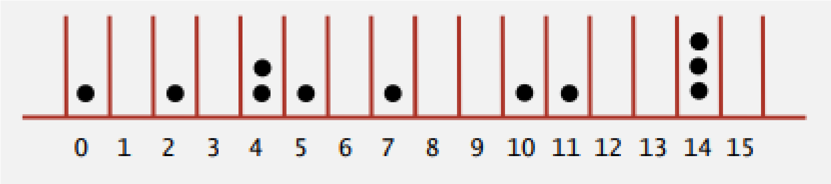
Hash functions for Strings

- It is common to use string-valued keys in hash tables
- What is a good hash function for strings?
- The basic approach is to use the characters in the string to compute an integer, and then take the integer mod the size of the table
- How to compute an integer from a string?
- You could just take the last two 16-bit chars of the string and form a 32-bit int
- But then all strings ending in the same 2 chars would hash to the same location; this could be very bad
- It would be better to have the hash function depend on all the chars in the string
- There is no recognized single "best" hash function for strings.

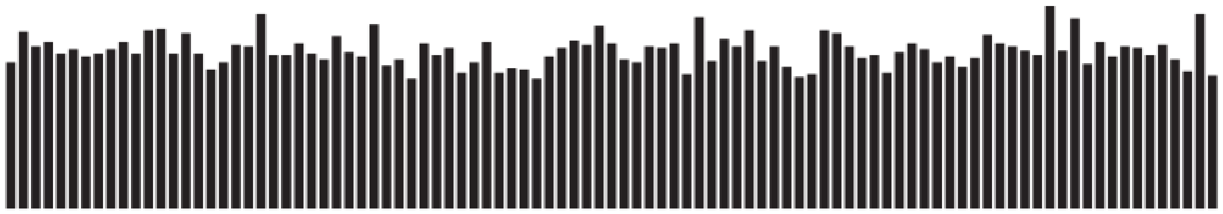
Uniform Hashing Assumption

Uniform hashing assumption. We assume that each key is ***equally likely*** to hash to an integer between 0 and $M - 1$.

Bins and balls. I.e., we assume that the hash function behaves like throwing balls uniformly at random into M bins.



Useful for predictions. Under the uniform hashing assumption we can prove things about average load factor, etc.

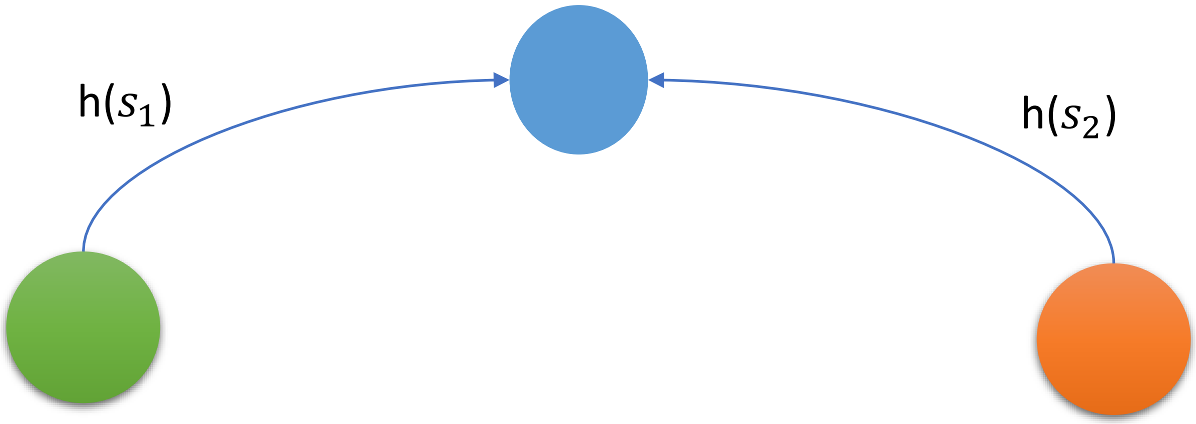


Hash value frequencies for words in Tale of Two Cities (M = 97)

Hash Function - Collisions

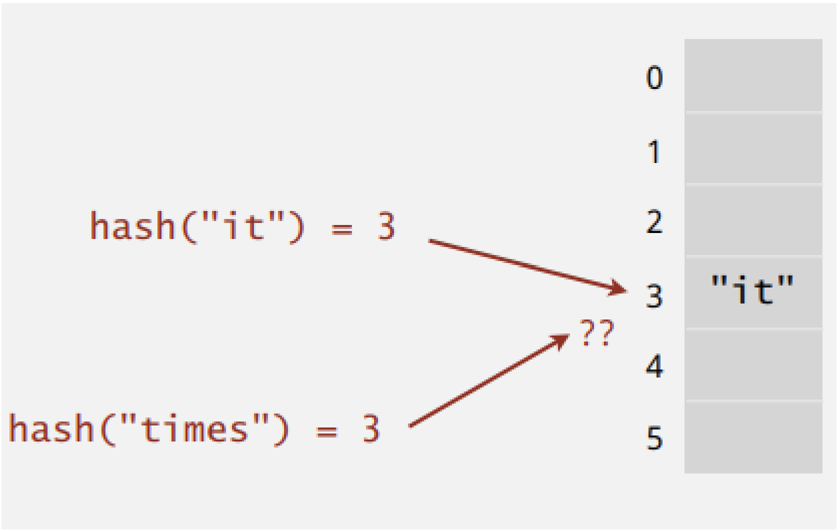
If h(s_1) = h(s_2) and $s_1 \neq s_2$

i.e., two different keys hashing to the same index



Example:

- Two people have the same birth year



Hash Tables - Example 1

(tomato, red)

hash of ‘tomato’ -> 26

26 mod 5 = 1

(spinach, green)

hash of ‘spinach’ -> 30

30 mod 5 = 0

(carrot, orange)

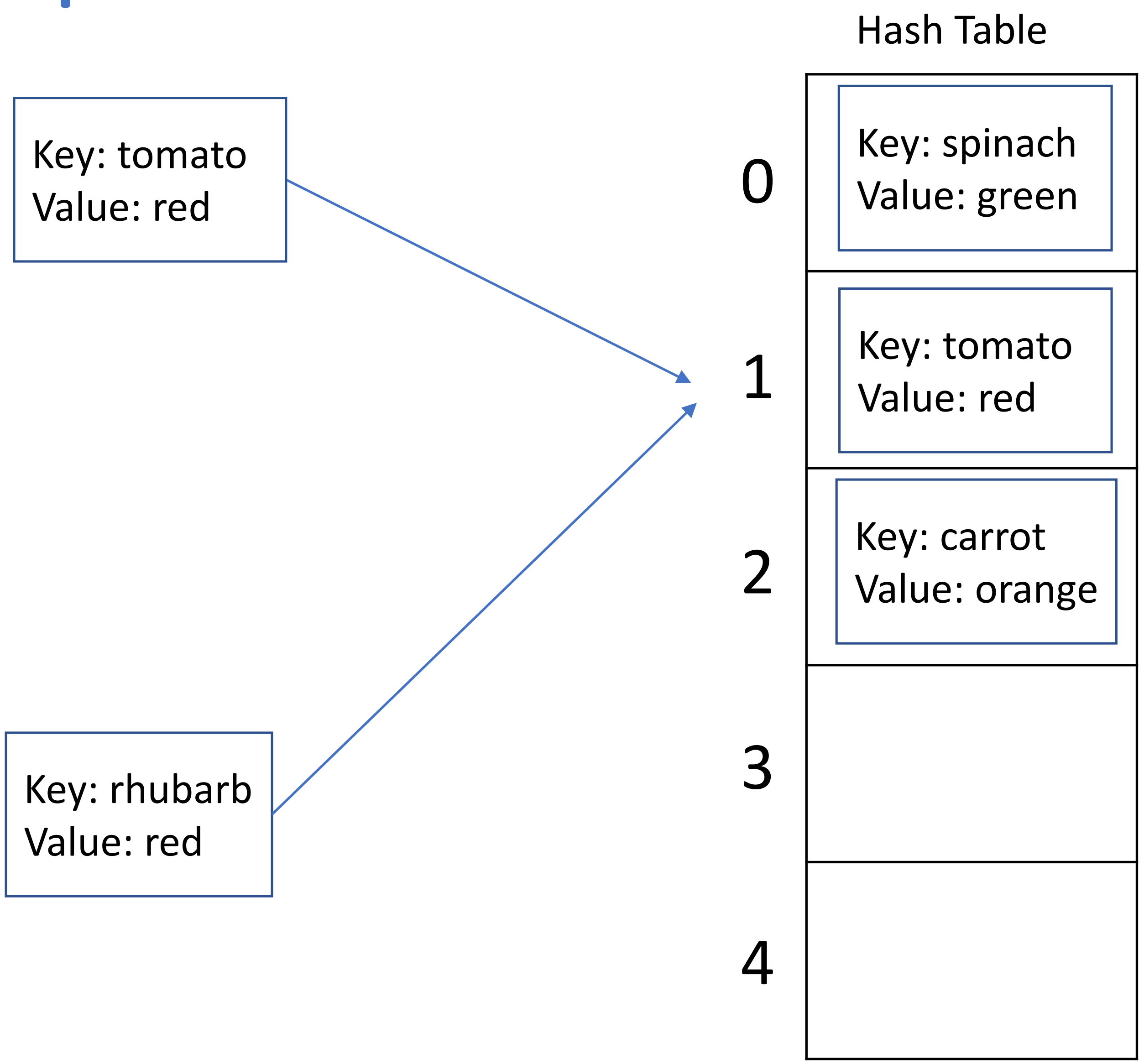
hash of ‘carrot’ -> 22

22 mod 5 = 2

(rhubarb, red)

hash of ‘rhubarb’ -> 36

36 mod 5 = 1



Hash Function - Collisions

Question: How do we deal with collisions efficiently?

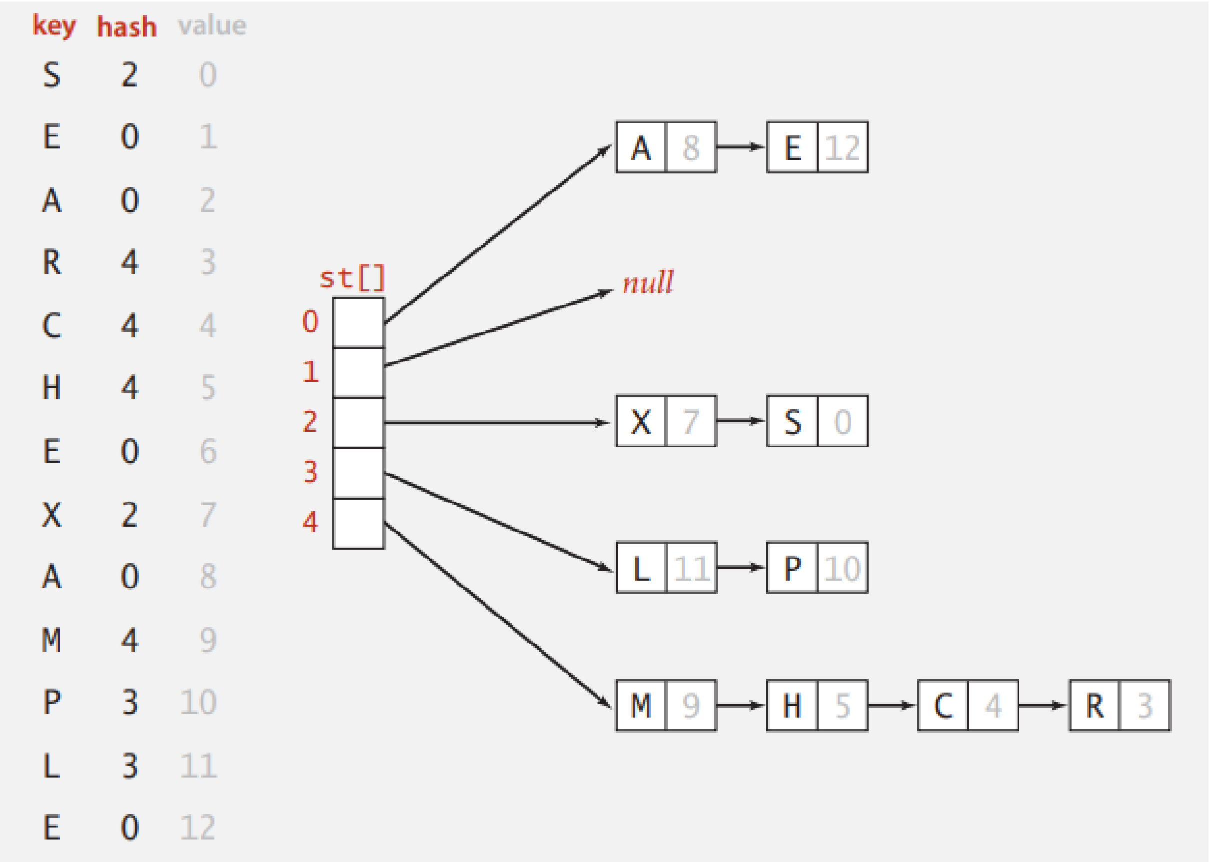
- Separate chaining
- Open addressing
  - Linear probing

Separate-chaining symbol table

Use an array of $M < N$ linked lists.

[H. P. Luhn, IBM 1953]

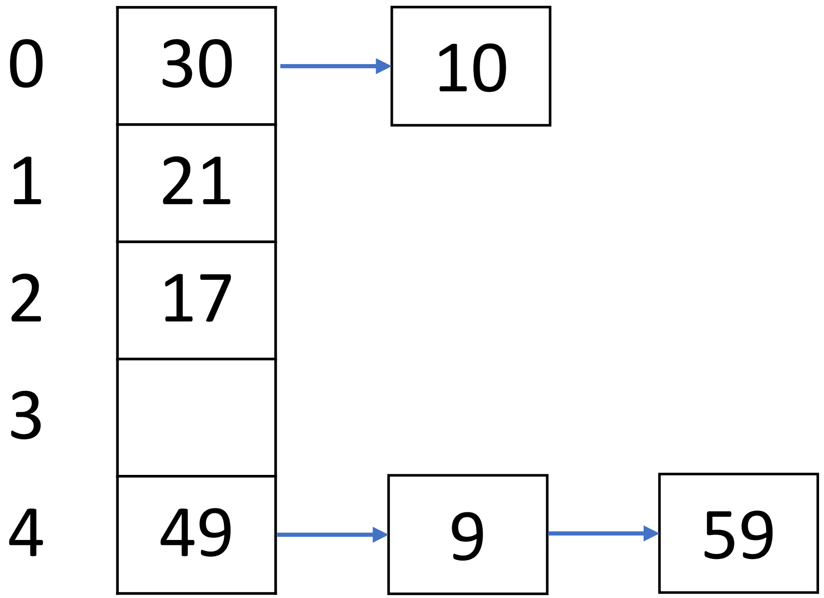
- Hash: map key to integer *i* between 0 and $M - 1$.
- Insert: put at front of the *i*th chain (if not already there).
- Search: we only need to search in the *i*th chain.



Chaining

h(x) -> x mod 5

x={ 21, 30, 48, 17, 9, 10, 59}



21 mod 5 = 1

30 mod 5 = 0

49 mod 5 = 4

17 mod 5 = 2

9 mod 5 = 4

10 mod 5 = 0

59 mod 5 = 4

Analysis Of Separate Chaining

Proposition. Under the *uniform hashing assumption*, there's a very high probability that the number of keys in a list is within a constant factor of *N* / *M*.

`equals()` and `hashCode()`

M times faster than sequential search

Consequence. Number of probes for search/insert is proportional to N / M.

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains become too long.
- Typical choice: select $M \sim N / 4$ $\Rightarrow$ constant-time operations (on average).

Question – Poll

Which statement is closer to the definition of load factor?

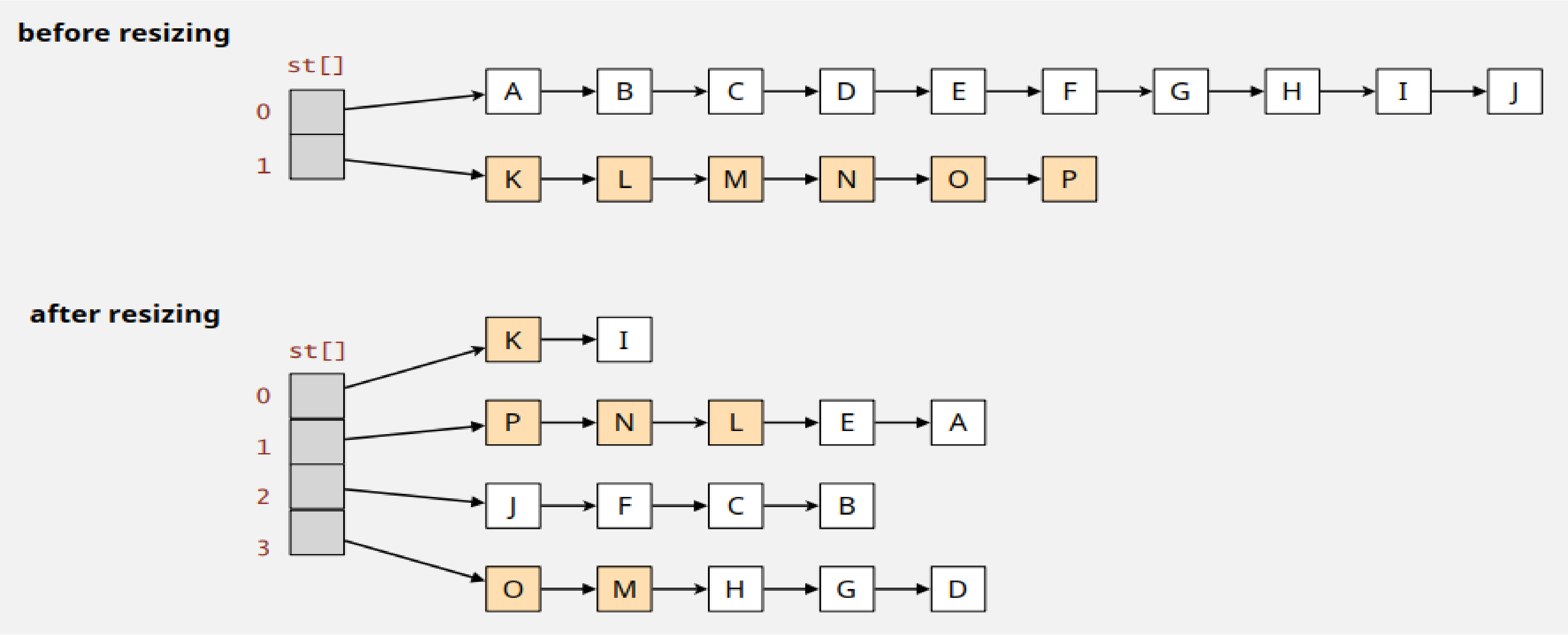
1) Average array size
2) Average key size
3) Average chain length
4) Average hash length

Resizing in a Separate-Chaining Hash Table

Goal. The average length of list is N / M = constant.

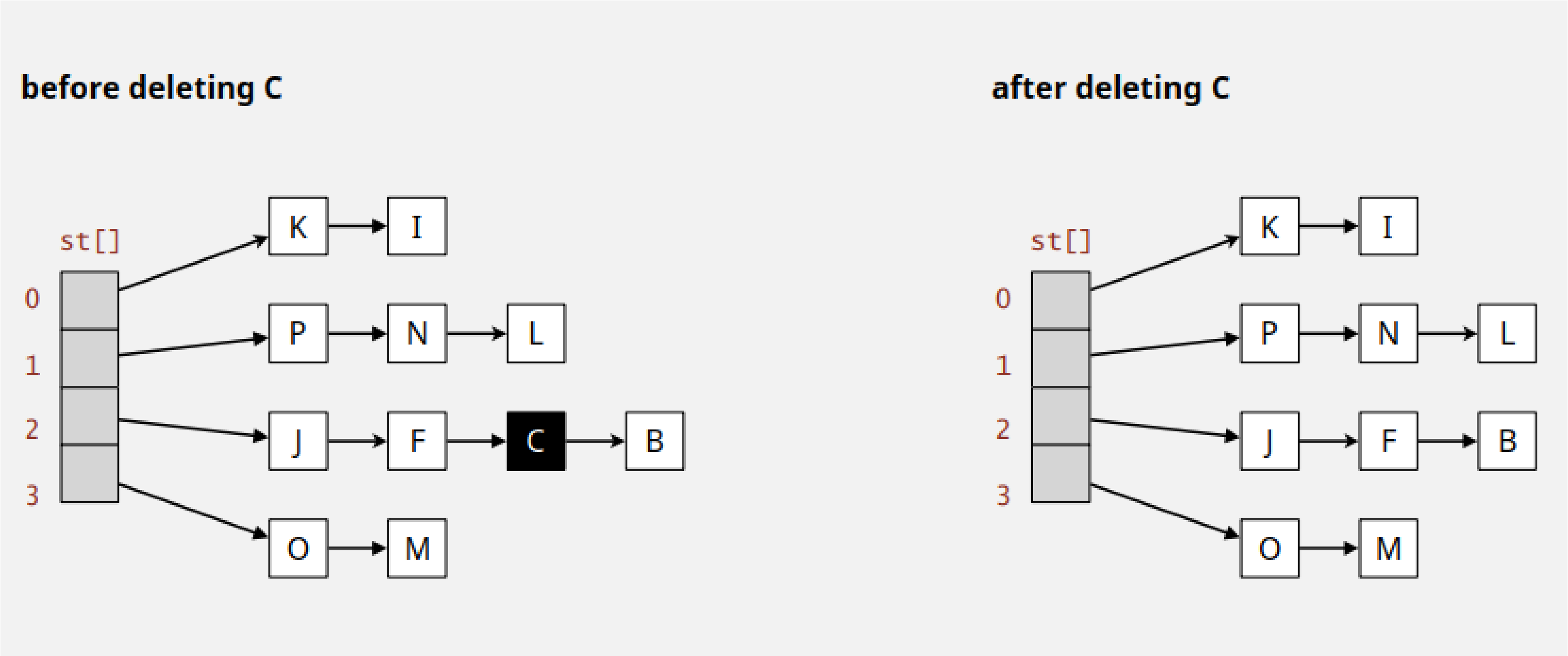
- Double the size of array M when $N / M \geq 8$.
- Halve the size of array M when $N / M \leq 2$.

Note. We need to rehash all keys when resizing.



Deletion in a Separate-Chaining Hash Table

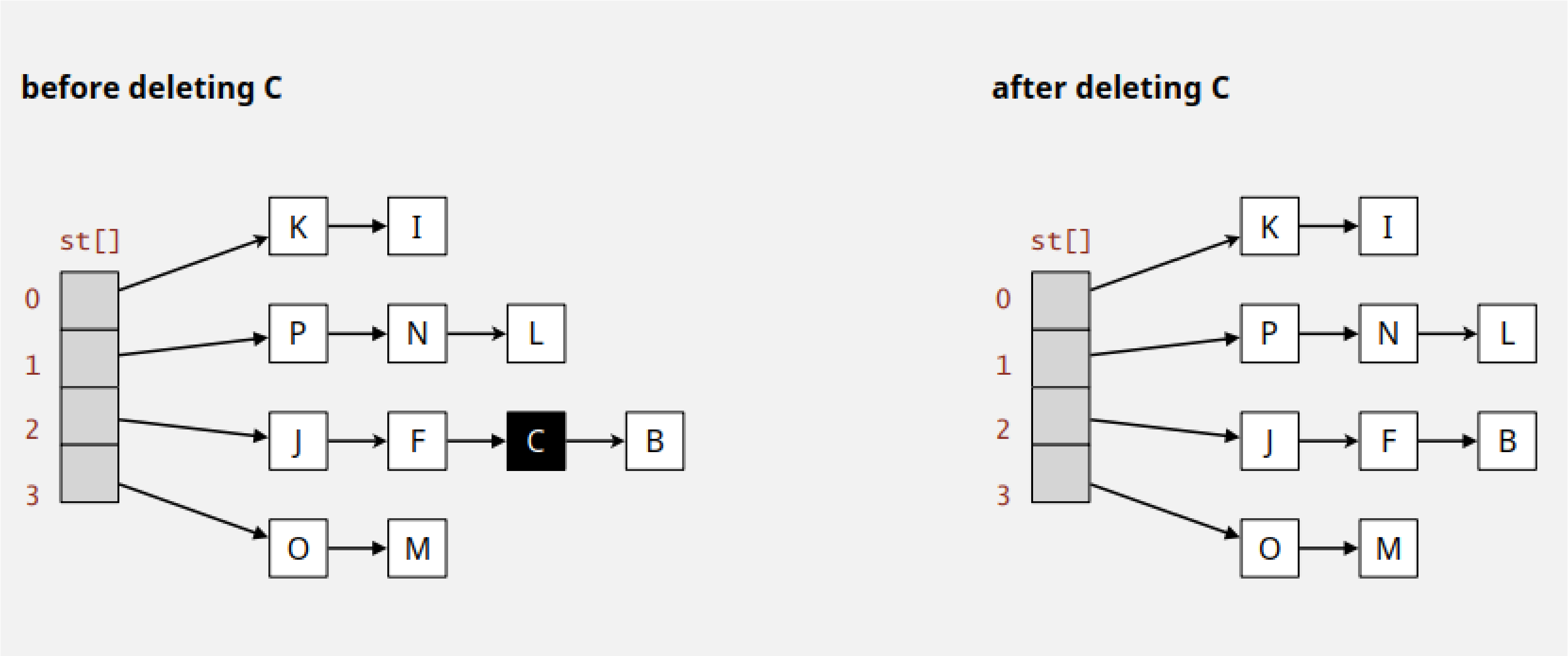
Q. How do we delete a key (and its associated value)? (Poll)



Deletion in a Separate-Chaining Hash Table

Q. How do we delete a key (and its associated value)? (Poll)

A. Easy: we need only consider the chain containing the key.



Collision resolution: Open Addressing

Open addressing. "Store N key-value pairs in a hash table of size M > N, relying on empty slots in the table to help with collision resolution" [1].

I.e., when a new key collides, find the next empty slot, and put it there [2].

Linear Probing is the simplest open-addressing method.

[1] Algorithm book by Sedgewick & Wayne, 2020.
[2] Amdahl-Boehme-Rocherster-Samuel, IBM 1953

Collision resolution: Linear Probing

Hash. Map key to integer *i* between *0* and *M-1*.

Insert. Put at table index *i* if free; if not try *i+1*, *i+2*, etc.

insert L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
M = 16							L									

Collision resolution: Linear Probing

Hash. Map key to integer *i* between *0* and *M-1*.

Insert. Put at table index *i* if free; if not try *i+1*, *i+2*, etc.

insert L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
M = 16									L							

Collision resolution: Linear Probing

Hash. Map key to integer *i* between *0* and *M-1*.

Insert. Put at table index *i* if free; if not try *i+1*, *i+2*, etc.

insert L
hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

Collision resolution: Linear Probing

Hash. Map key to integer *i* between *0* and *M-1*.

Insert. Put at table index *i* if free; if not try *i+1*, *i+2*, etc.

Search. Search table index *i*; if occupied but no match, try *i+1, i+2*, etc.

search H
hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16					H											

Collision resolution: Linear Probing

Hash. Map key to integer *i* between *0* and *M-1*.

Insert. Put at table index *i* if free; if not try *i+1*, *i+2*, etc.

Search. Search table index *i*; if occupied but no match, try *i+1, i+2*, etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`st[]`	P	M			A	C	S	H	L		E				R	X
`M = 16`								H								

search hit
(return corresponding value)

Collision resolution: Linear Probing

Search. Search table index *i*; if occupied but no match, try *i+1, i+2*, etc.

Possible outcome of search.

- key equal to the search key: search hit
- Null key at the indexed position, empty position: search miss
- Key not equal to the search key: try the next key

Note. Array size M must be greater than number of key-value pairs N

Resizing in a linear-probing hash table

Goal. Average length of list N / $M \le 1/2$.

- Double size of array M when N / $M \ge 1/2$.
- Halve size of array M when N / $M \le 1/8$.

Note. We need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

after resizing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]					A		S				E				R	
vals[]					2		0				1				3	

Deletion in a linear-probing hash table

Q. How do we delete a key (and its associated value)?

A. Requires some care: we can't just delete array entries.

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

doesn't work, e.g., if hash(H) = 4

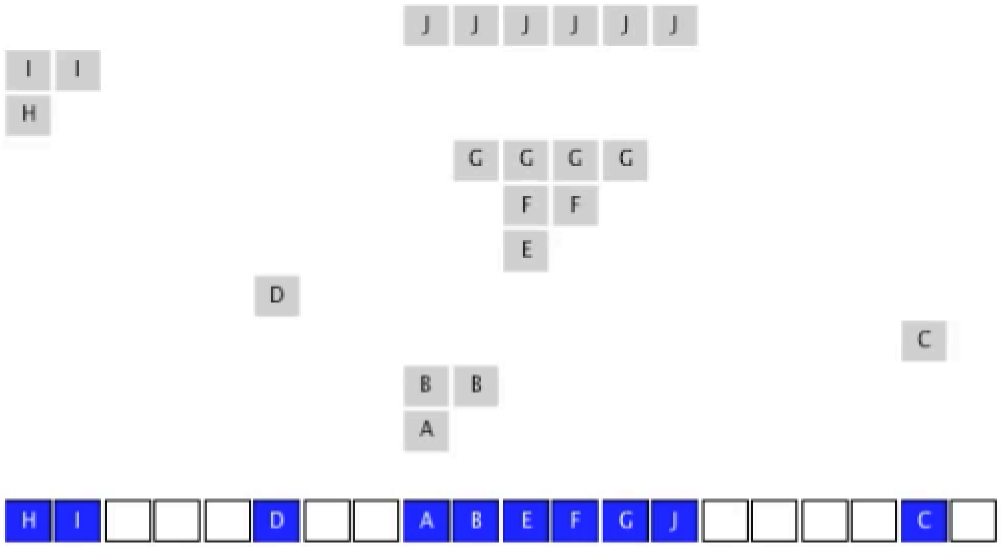
after deleting S ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7

Problem with Linear Probing: Clustering

Cluster. A contiguous block of items.

Observation. New keys likely to hash into middle of big clusters



Map - Methods

Map from objects to other objects:

- Filename -> file on disk
- Student ID -> student name
- Contact name -> phone number

Map from set of objects to the set of values is a data structure with following methods

HasKey(OBJEKT)

```
HasKey(OBJEKT)
 L <- myArray[h(OBJEKT)]
 for (Ob,val) in L:
     if Ob==OBJEKT:
         return True
return False
```

Map - Methods

Get(OBJEKT)

```
Get(OBJEKT)
 L <- myArray[h(OBJEKT)]
 for (Ob,val) in L:
      if Ob==OBJEKT:
           return val
return SpecialVal
```

Map - Methods

```
Set(OBJEKT, Value)
```

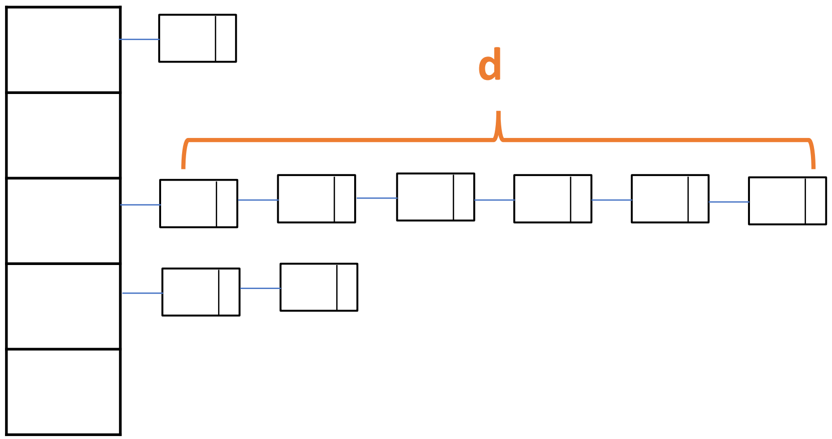
```
Set(OBJEKT, VALUE)
 L <- myArray[h(OBJEKT)]
 for pair in L:
     if pair.OBJEKT==OBJEKT:
         pair.VALUE <- VALUE
         return
L.append(OBJEKT, VALUE)
```

Method Analysis – Running Time

Suppose *d* is the length of the longest list in the hash table.

Then the running time of `HasKey()`,`Get()` and `Set()` depend on the size of the longest linked list in the array.

- If length(L)=*d*, then we need to go through all the array
- If length(L)=0, we need to check that as well, i.e., a constant time
- Small *d* is desirable



Method Analysis – Memory Consumption

Suppose n be the size of different keys and m is the cardinality of hash function. Then the memory consumption for chaining will be *n+m*

- Store *n* keys
- Store an array of size *m*
- Small n and m is desirable

Hash Tables

Set:

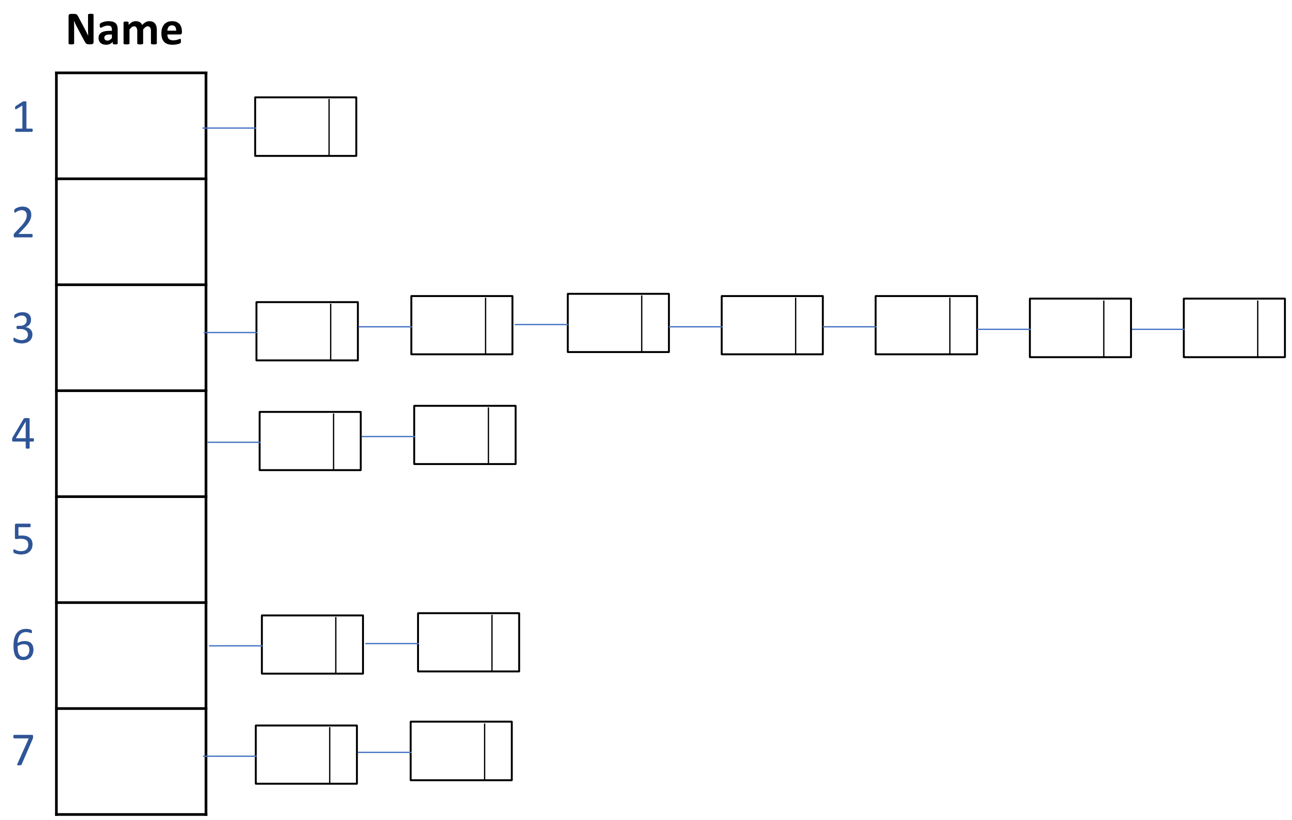
- `Unordered_set` in C++
- `HashSet` in Java
- `Set` in Python

Map:

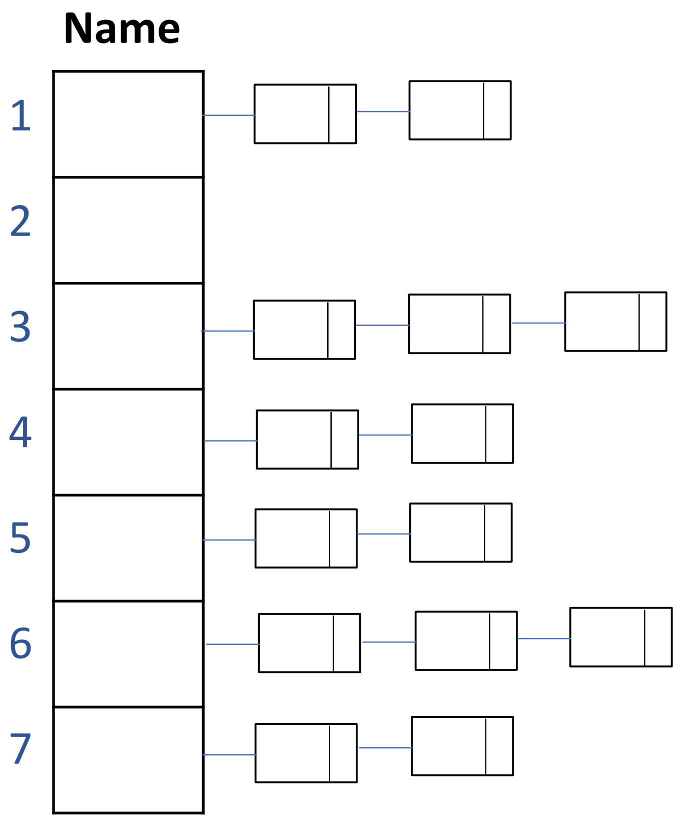
- `Unordered_map` in C++
- `HashMap` in Java
- `dic` in Python

Question – Analysis (Poll)

Which of the hash tables is a good example?



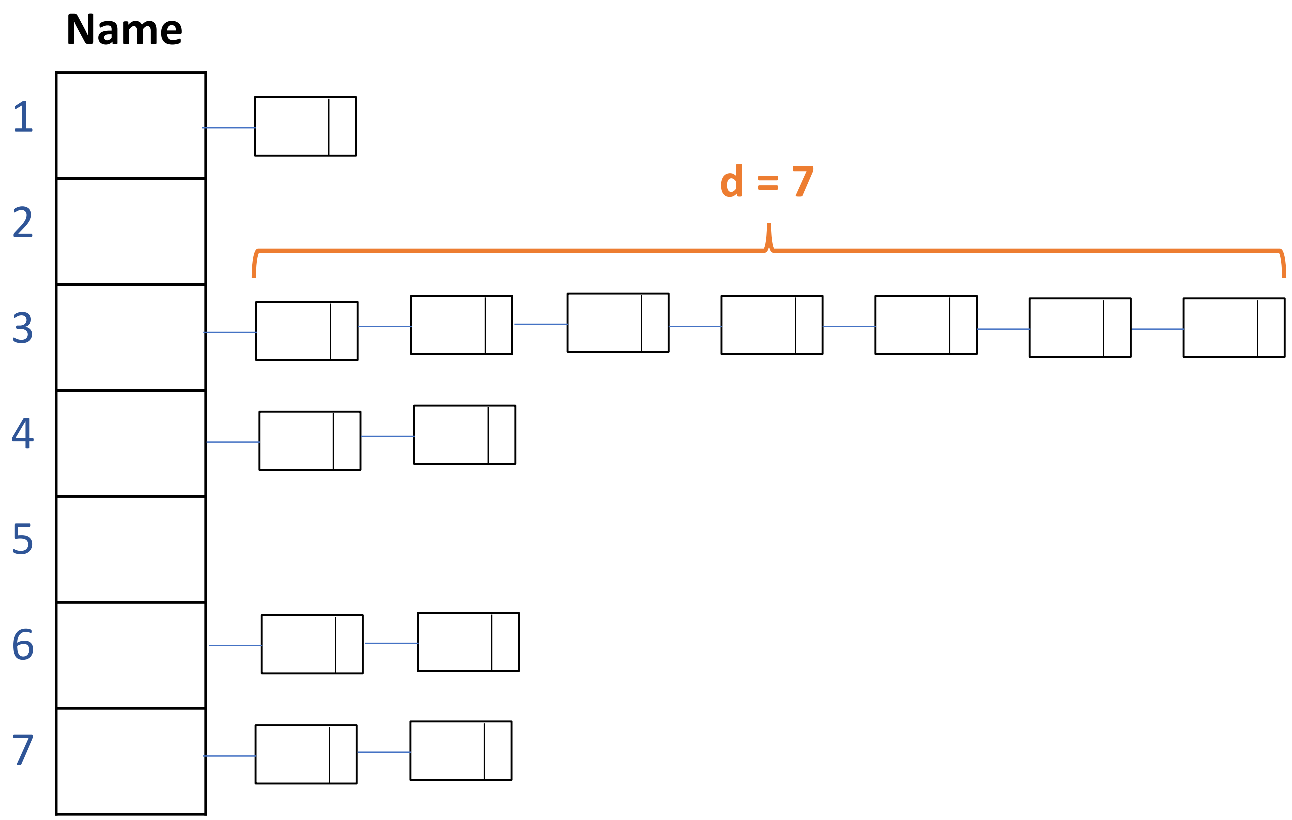
(1)



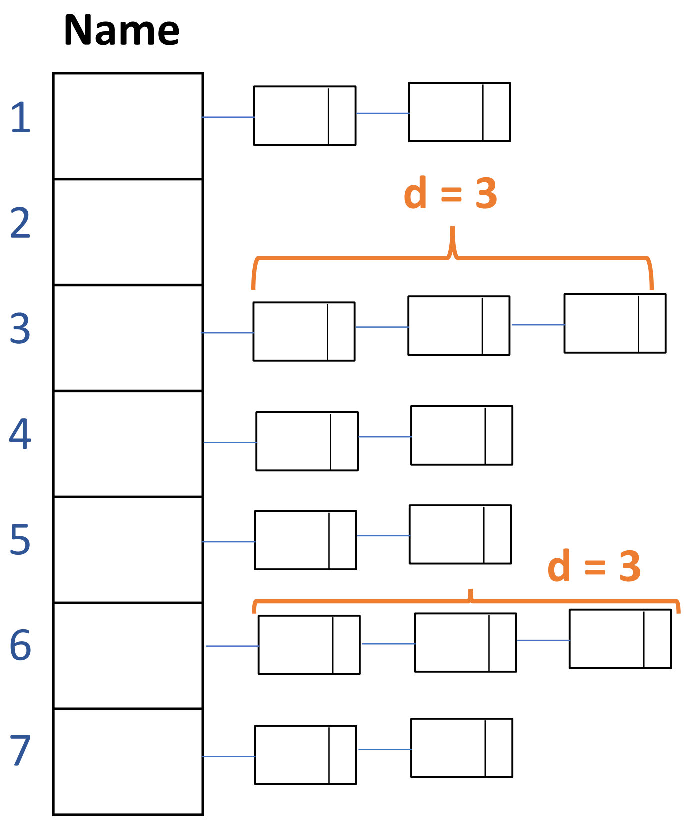
(2)

Question – Analysis (Poll)

Which of the hash tables is a good example?



(1)



(2)

Dynamic Hash Tables

For unknown number of keys:

- Use a large hash table!
  - Consequence: waste of memory

What to do?

- Dynamic allocation:
  - Resize the hash table when needed
  - Create a new hash table and rehash

Rehash analysis

- $O(n)$
- Rare occurrence

Applications

Pattern Search in Text. Find all occurrence of pattern P in text T

T: Twitter, articles, News websites

P: a special phrase, a word or a sentence

Other Example:

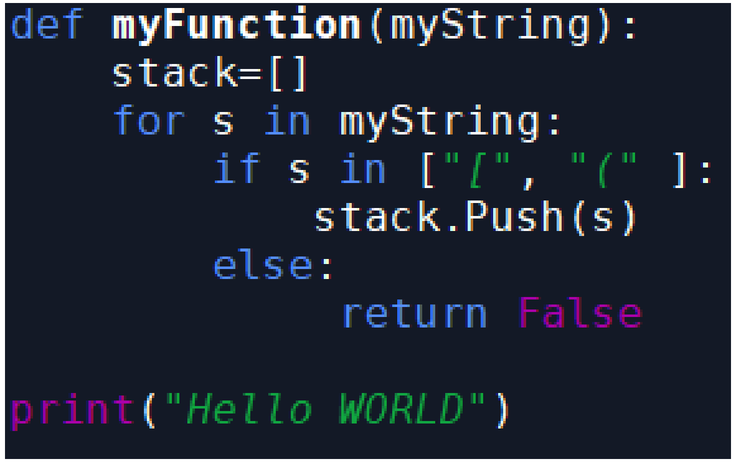
- Find reviews about a product

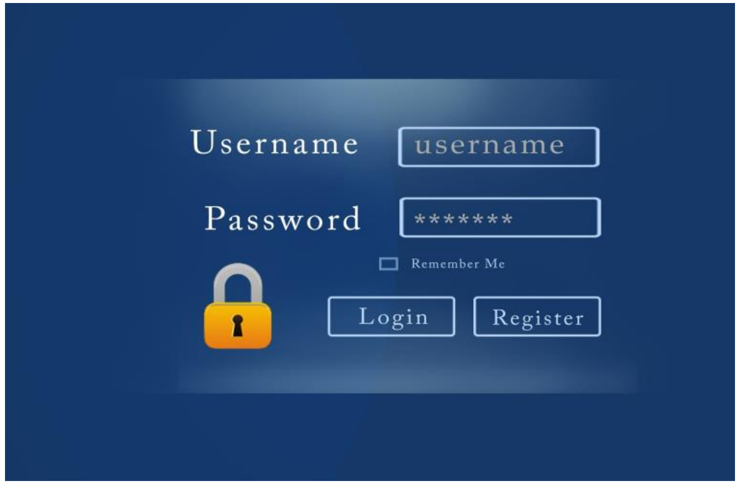
Applications

- Keywords in programming languages

  Examples: double, if, float, while, for

- File systems
- Password verification
- Cloud services
  - Google Drive, DropBox





Applications

User_1 ->

User_2 ->

.

.

.

User_k ->

Applications

User_1 ->

User_2 ->

.
.
.

User_k ->

Question – Poll

Which data structures support ordered operations *efficiently*?

1) unsorted arrays and sorted arrays
2) sorted arrays and binary search trees
3) hash tables and binary search trees
4) hash tables and sorted arrays

Hash tables are not ordered!

Ordered operations. Many operations require some underlying order, e.g.:

- Ordered traversal.
- Finding the minimum / maximum.
- Finding the floor / ceiling.
- Selecting the *k*th element / finding the *k* for a given element.

Data structures support ordered operations *efficiently*?

- **BSTs** and **sorted arrays**: Logarithmic time or better.
- **Hash tables** and **unsorted arrays**: Nope

Summary

- No random hash values that we cannot retrieve
- Balanced hash table
  - d and M small, few collisions and not very large hash table
- Operations run fast
- Fast calculation of the hash values
- Is there any universal hash functions?

A. No

Important point:

If the number of keys is much larger than the size of hash table, then many collisions can happen!

Hash tables vs. balanced search trees

Hash tables.

- The only effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus log N compares).
- For complex keys (e.g., strings or arrays), the hash code can be cached.

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered symbol table operations (ordered traversal etc.)
- Easier to implement compareTo() correctly than equals()

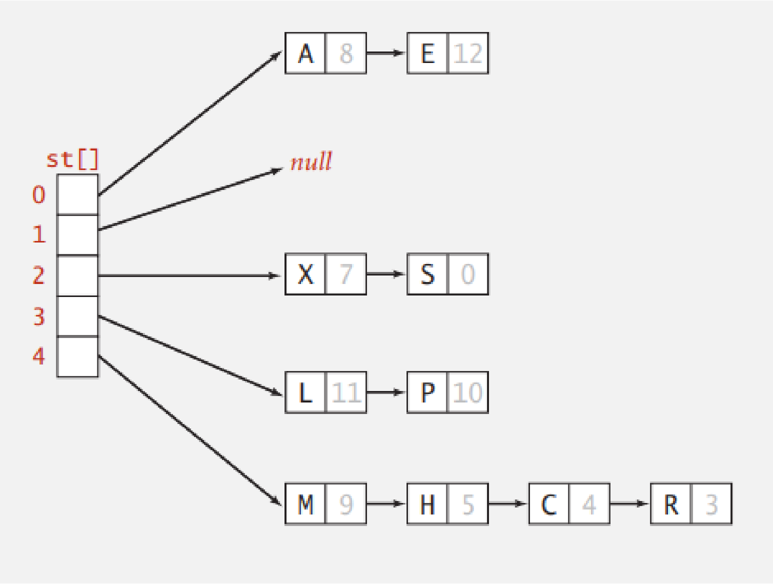
Separate chaining vs. linear probing

Separate chaining.

- Performance degrades gracefully.
- Clustering is less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

Summary

Implementation	Worst-case cost			Average case cost (after N random inserts)		memory
	search	insert	delete	Search hit	insert	
Sequential search (unordered list)	*O(N)*	*O(N)*	*O(N)*	*O(N)*	*O*(N)	*O*(N)
Binary search (ordered array)	*O(Log N)*	*O(N)*	*O(N)*	*O(Log N)*	*O*(N)	*O*(N)
BST	*O(N)*	*O(N)*	*O(N)*	*O*(log N)	*O*(log N)	*O*(N)
Separate chaining	*O(N)*	*O(N)*	*O(N)*	*O*(1) *	*O*(1) *	*O*(N + M)
Linear probing	*O(N)*	*O(N)*	*O(N)*	*O*(1) *	*O*(1) *	*O*(N)

* under the uniform hashing assumption

Symbol table implementations: Summary

Implementation	Worst-case cost			Average case cost (after N random inserts)		memory
	search	insert	delete	Search hit	insert	
Sequential search (unordered list)	*N*	*N*	*N*	*½ N*	*N*	*48N*
Binary search (ordered array)	*Log N*	*N*	*N*	*Log N*	*½ N*	*16N*
BST	*N*	*N*	*N*	*1.4 log N*	*1.4 log N*	*64N*
Separate chaining	*N*	*N*	*N*	3-5 *	3-5*	*64N+32M*
Linear probing	*N*	*N*	*N*	3-5*	3-5*	*32N to 128N*

* under the uniform hashing assumption