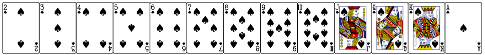
COMP 2011: Data Structures

Lecture 12. Collision Handling and Conclusions

Dr. CAO Yixin

November, 2021



Think GREEN
Only print if it's essential

Review

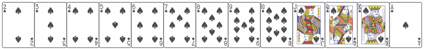
- A hash function should spread out the data evenly among the entries in the table.
- More formally: Any key should be equally likely to hash to any of the locations.
- The probability distribution on the keys is usually not known.

Review question

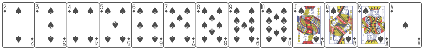
Which of the following hashing strategies are good?

- $a.b$ for IP addresses
- $b.c$ for IP addresses
- $c.d$ for IP addresses

Can you propose a better one?



Collisions

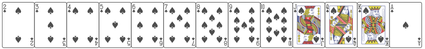


Subject codes of computing (part)

1001, 1003, 1011, 1901, 2011, 2012, 2021, 2022, 2121, 2222, 2322, 2411, 2421, 2422, 2432, 3011, 3021, 3121, 3122, 3131, 3133, 3134, 3211, 3222, 3233, 3235, 3334, 3335, 3421, 3422, 3432, 3438, 3511, 3512, 3531, 3911, 3921, 4000, 4001, 4121, 4122, 4123, 4125, 4127, 4133, 4134, 4135, 4141, 4142, 4146, 4322, 4332, 4334, 4342, 4422, 4431, 4433, 4434, 4435, 4438, 4441, 4442, 4512, 4531, 4911, 4913, 4921

$$\Big\downarrow \%97$$

31, 33, 41, 58, 71, 72, 81, 82, 84, 88, 91, 83, 93, 94, 7, 4, 14, 17, 18, 27, 29, 30, 10, 21, 32, 34, 36, 37, 26, 27, 37, 43, 19, 20, 39, 31, 41, 23, 24, 47, 48, 49, 51, 53, 59, 60, 61, 67, 68, 72, 54, 64, 66, 74, 57, 66, 68, 69, 70, 73, 76, 77, 50, 69, 61, 63, 71



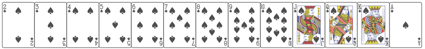
Collisions

Collision: $k_1 \neq k_2$ but $h(k_1) = h(k_2)$ e.g., $57\%13 = 31\%13 = 5$

True or false

[ ] `"Siblings".hashCode() == "Teheran".hashCode`

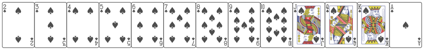
[ ] `"Ca".hashCode() == "DB".hashCode`



Collisions are inevitable

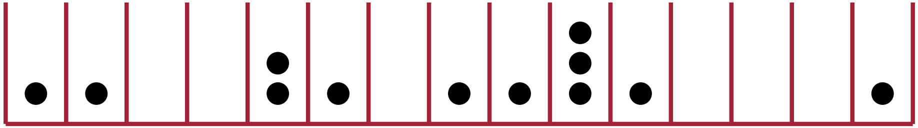
- A good hash function minimizes the chance of collisions.
- To completely avoid collisions, the *only* way is to make M____N.
- To avoid collisions with high probability, we need to make $M \geq$ ________.

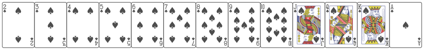
Conclusion: Any reasonable hash function has collisions. We've to live with it.



The chance

In a table of size M, how many elements we can put with $< \frac{1}{2}$ chance of collision?

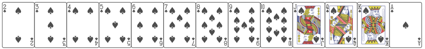




The birthday paradox ( )

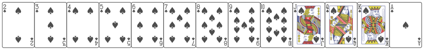
- There are about 40 professors in the Department of Computing.
- Are there two of them having the same birthday (not necessarily the same year)?
- The probability p that all of us have different birthdays is

  A. $p > 0.5$;
  B. $0.1 \leq p < 0.5$;
  C. $0.01 \leq p < 0.1$;
  D. $p < 0.01$;

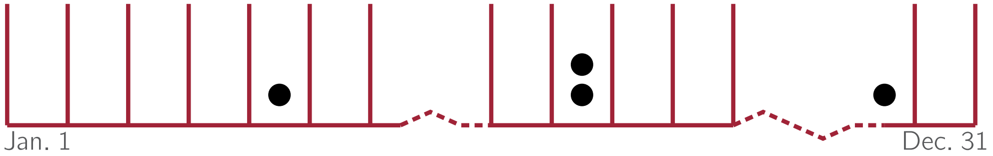


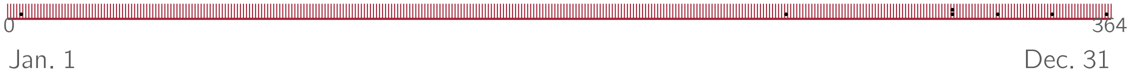
The birthday paradox ( )

- There are about 40 professors in the Department of Computing.
- Are there two of them having the same birthday (not necessarily the same year)?
- The probability p that all of us have different birthdays is
  - A. $p > 0.5$;
  - B. $0.1 \leq p < 0.5$;
  - C. $0.01 \leq p < 0.1$;
  - D. $p < 0.01$;
- The chance is $> 50\%$ among 23 randomly selected persons $23 \approx \sqrt{\frac{365\pi}{2}}$

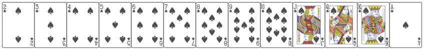


The conclusion





- Expect two in the same slot after about $\sqrt{\frac{\pi M}{2}}$ elements.
- To store n elements, how large M you need to avoid "two in the same"?

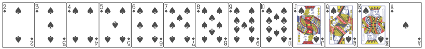


Load factor

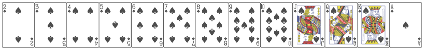
$N \gg n$ and $N \gg M$, but the relation between n and M have not been specified.

The *load factor* of a hash table, which measures how full the table is:

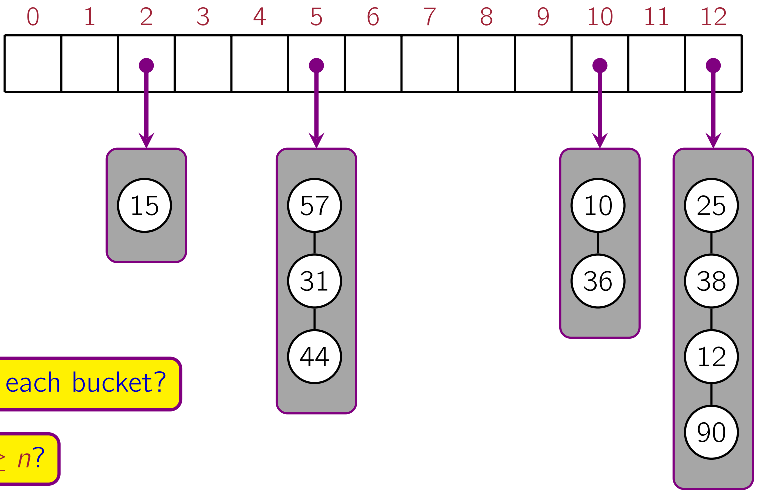
$$\lambda = \frac{n}{M}$$



Separate Chaining



Illustration



How to organize each bucket?

Do we need $M \geq n$?

$M = 13$ and $h(k) = k \,\%\, 13$.

An interactive demo

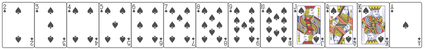
Average case analysis of separate chaining

For hash tables with separate chaining.

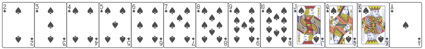
- With separate chaining, λ is the average length of each linked list.
- Experiments and average-case analyses suggest that we should maintain $\lambda < 0.9$.
- By default, Java uses $\lambda < 0.75$.

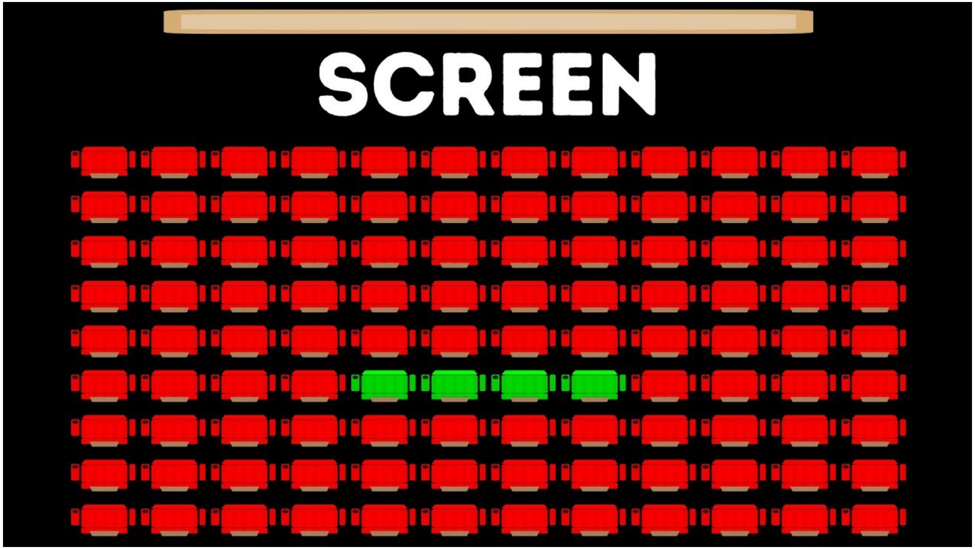
- A major disadvantage: It wastes a lot of space.
- You can change the implementation such that an auxiliary data structure is only required when there are collisions, but it becomes more complicated.

Every data structure is a combination of arrays and linked lists.



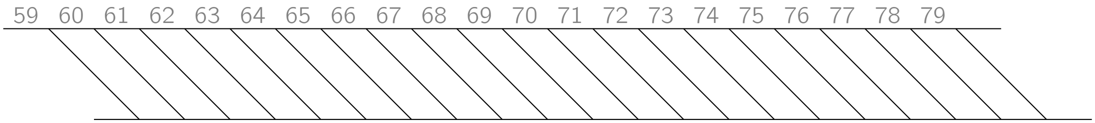
Open Addressing





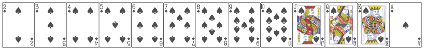
When you enter a theater, the best seats have been taken, then...

Handling subject codes with open addressing

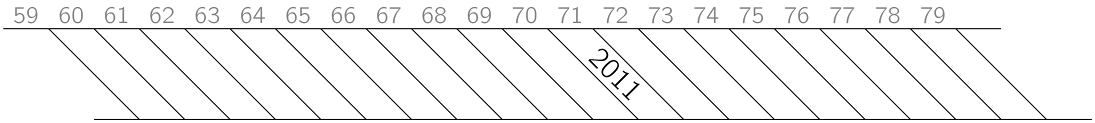


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

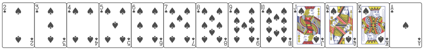


Handling subject codes with open addressing

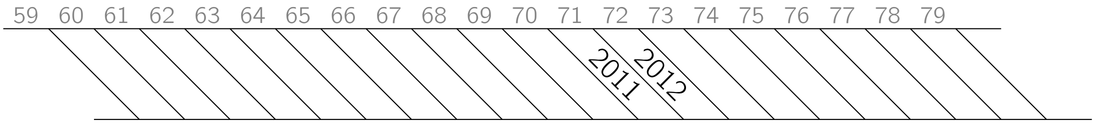


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

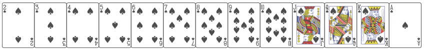


Handling subject codes with open addressing

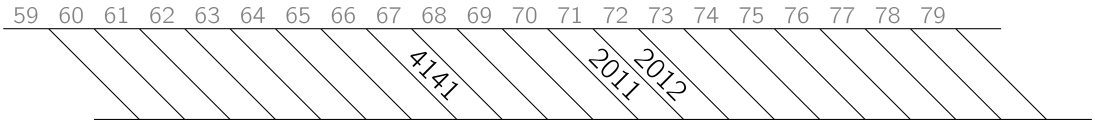


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72),
4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69),
4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

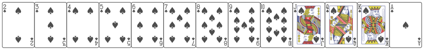


Handling subject codes with open addressing

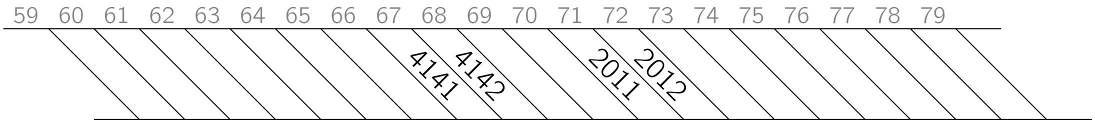


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

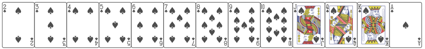


Handling subject codes with open addressing

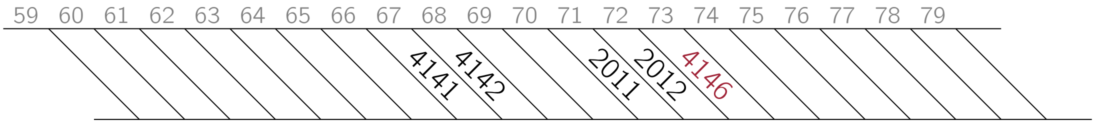


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

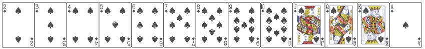


Handling subject codes with open addressing

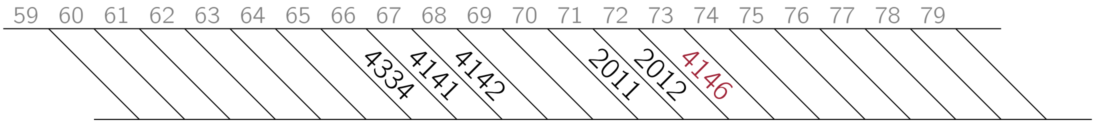


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72),
4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69),
4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

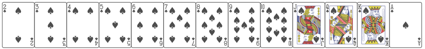


Handling subject codes with open addressing

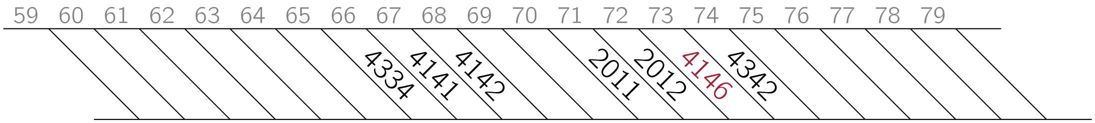


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

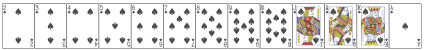


Handling subject codes with open addressing

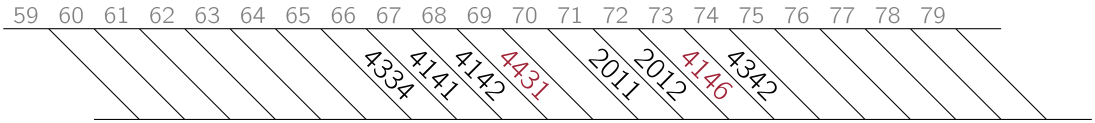


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

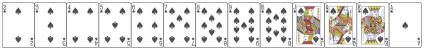


Handling subject codes with open addressing

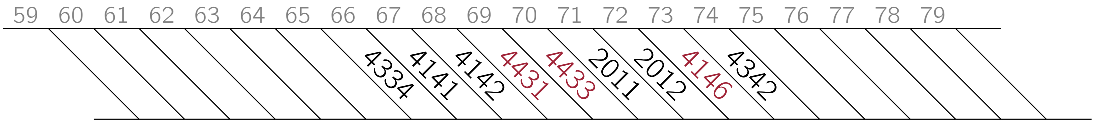


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

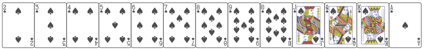


Handling subject codes with open addressing

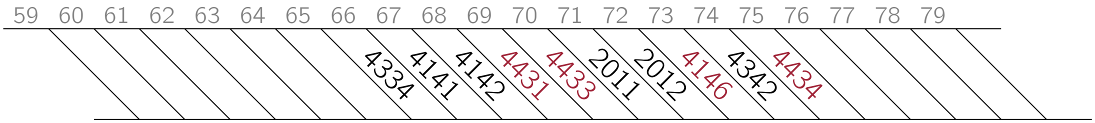


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

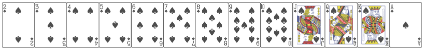


Handling subject codes with open addressing

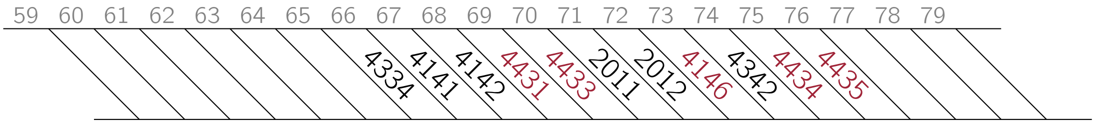


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

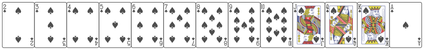


Handling subject codes with open addressing

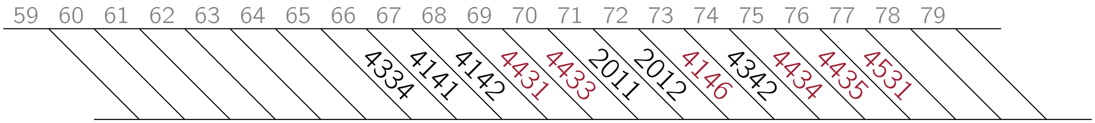


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

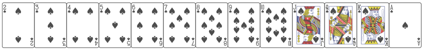


Handling subject codes with open addressing

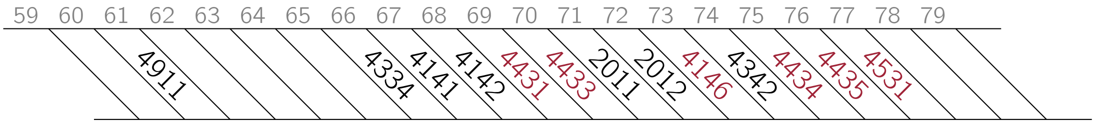


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

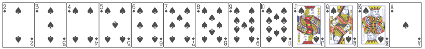


Handling subject codes with open addressing

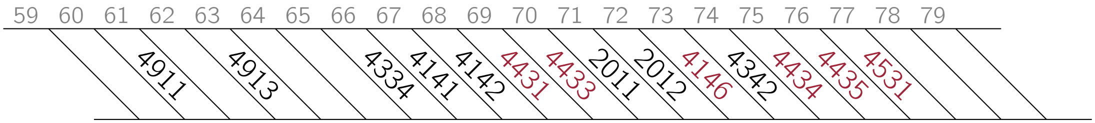


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72),
4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69),
4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

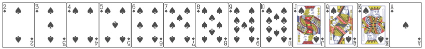


Handling subject codes with open addressing

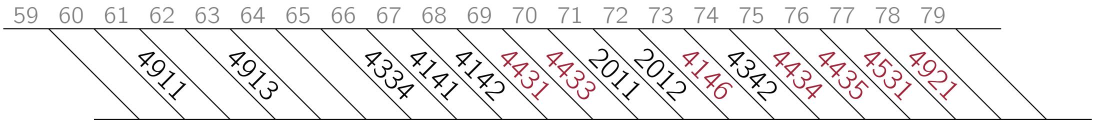


$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

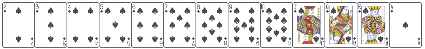


Handling subject codes with open addressing



$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)

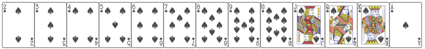


Open addressing ( )

- All data are stored in the array, so $\lambda \leq 1$. $M \geq n$
- It's better to keep $\lambda < 0.5$;
  otherwise large clusters form and the performance degrades quickly.

Suppose $i = h(k)$.

- Insert: Put at table index i if available; otherwise, try $i+1$, $i+2$, ...
- Search. Search table index i; if occupied but no match, try $i+1$, $i+2$, ...,
  until an unoccupied slot.
- Delete. How?

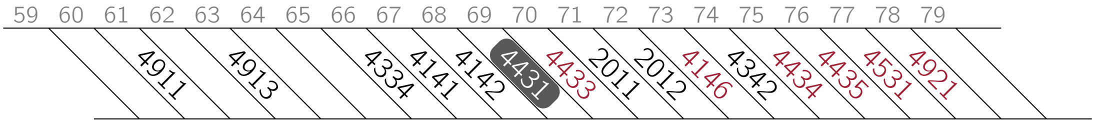


Deletions (the simple approach)

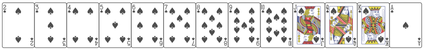
$h(k) = k \bmod 97$

We need to delete 4431.

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)



- It's not sufficient to simply delete 4431: searching 4433 will fail even it exists.
- We need to move 4433 to slot 69, but not 2011.
- Are we done? How about 4434?



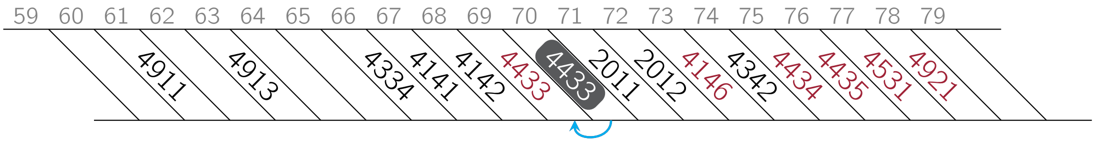
An interactive demo

Deletions (the simple approach)

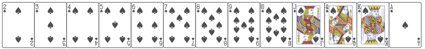
$h(k) = k \bmod 97$

We need to delete 4431.

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)



- It's not sufficient to simply delete 4431: searching 4433 will fail even it exists.
- We need to move 4433 to slot 69, but not 2011.
- Are we done? How about 4434?



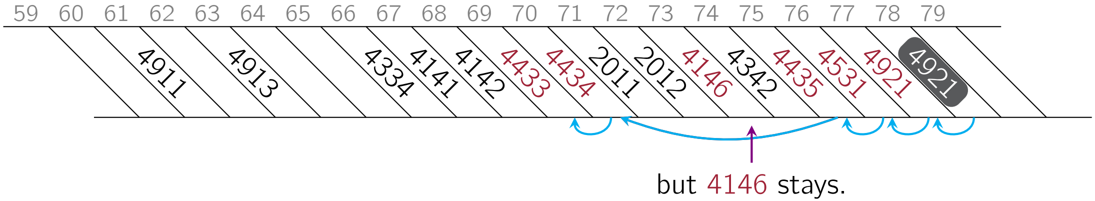
An interactive demo

Deletions (the simple approach)

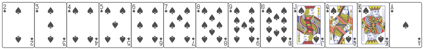
$h(k) = k \bmod 97$

We need to delete 4431.

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61), 4913 (63), 4921 (71)



- It's not sufficient to simply delete 4431: searching 4433 will fail even it exists.
- We need to move 4433 to slot 69, but not 2011.
- Are we done? How about 4434?



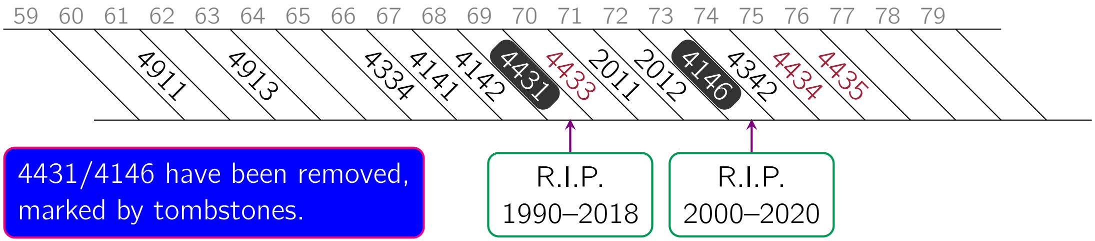
An interactive demo

How about deleting 4146?

Deletions (the tombstone approach)

$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61)



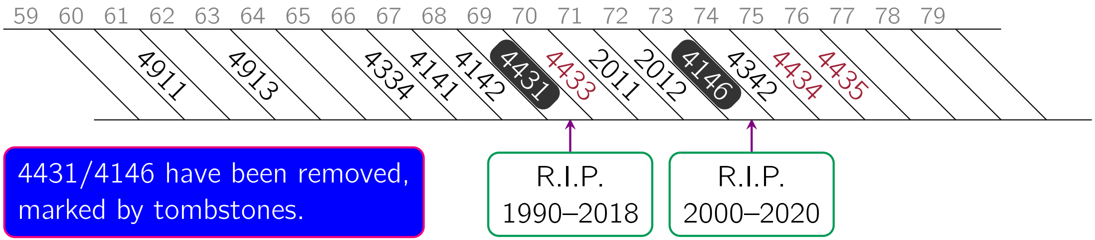
4431/4146 have been removed, marked by tombstones.

- Delete: establish a tombstone to mark this position *was* occupied.
- Insert: a slot with a tombstone is available.
- Search: stop only at an unoccupied slot without a tombstone.

Deletions (the tombstone approach)

$h(k) = k \bmod 97$

2011 (71), 2012 (72), 4141 (67), 4142 (68), 4146 (72), 4334 (66), 4342 (74), 4431 (66), 4433 (68), 4434 (69), 4435 (70), 4531 (69), 4911 (61)



4431/4146 have been removed, marked by tombstones.

The slots can be reused, e.g., by 4913 (63) and 4921 (71).

- Delete: establish a tombstone to mark this position *was* occupied.
- Insert: a slot with a tombstone is available.
- Search: stop only at an unoccupied slot without a tombstone.

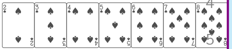
The codes

$t[i]$: tombstone at i.

```
void insert(int k) {
    int i = h(k);
    for (; (!t[i]) && data[i]!=-1; i=(i+1)%M)
        if (data[i] == k) break;
    data[i] = k;
}
```

```
void delete(int k) {
    for(int i=h(k); t[i]||data[i]!=-1; i=(i+1)%M)
        if ((!t[i]) && key == data[i]) {
            t[i] = true;
            data[i] = -1;
            size--;
        }
}
```

```
int search(int k) {
    for(int i=h(k); t[i]||data[i]!=-1; i=(i+1)%M)
        if ((!t[i]) && key == data[i]) return i;
    return -1;
}
```



Quadratic probing

$$h(k) \to (h(k) + 1^2)\%M \to (h(k) + 2^2)\%M \to (h(k) + 3^2)\%M \to \cdots$$

It does avoid the kinds of clustering patterns that occur with linear probing, but it creates its own kind of clustering, called secondary clustering.

Demo

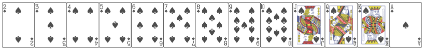
Double hashing

$$h(k) \to (h(k) + h'(k))\%M \to (h(k) + 2h'(k))\%M \to (h(k) + 3h'(k))\%M \to \cdots,$$

where h' is a secondary hash function.

A common choice is $h'(k) = q - (k\%q)$, for some prime number $q < M$.

Demo



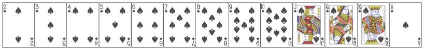
Exercises

We use an 11-entry hash table, with $h(i) = (3i + 5) \mod 11$ to hash the keys. Write down the results after inserting

$$12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5.$$

1. assuming collisions are handled by separate chaining;
2. assuming collisions are handled by linear probing.

3. assuming collisions are handled by quadratic probing;
4. assuming collisions are handled by double hashing using the secondary hash function $h'(k) = 7 - (k \mod 7)$?

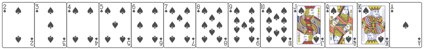


Efficiency

	Hash tables		arrays	
	average-case	worst-case	unsorted	sorted
insert	$O(1)$	$O(n)$	$O(1)$	$O(n)$
delete	$O(1)$	$O(n)$	$O(n)$	$O(n)$
search	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$

- Hash tables with separate chaining and hash tables with open addressing have the same average- and worst-case running time.
- But there are subtle differences.
- Separate chaining is easier to implement, less sensitive to load factor.
- Open addressing is more efficient, but degrade quickly with $\lambda \to 1$.

More information

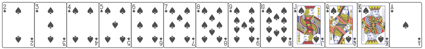


Summary

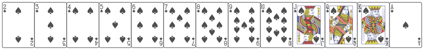
- The hashing of a key to an already-occupied array cell is called a collision.
- Two major ways to handle collisions: separate chaining and open addressing.
- In separate chaining, each array element consists of a linked list.
- In open addressing, an item doesn't need to be put at index of its hash value.
- The load factor is the ratio of data items to the array size. $\lambda = \frac{n}{M}$
- Good practices: $\lambda < 0.75$ (separate chaining) and $\lambda < 0.5$ (open addressing).
- In linear probing,
  - A cluster is a contiguous sequence of occupied slots.
  - Sizes of clusters increase with load factor.
  - Large clusters need more steps for each operation.
- Alternative strategies of open addressing: quadratic probing and double hashing.

Thorup and Zhang:

"Hash tables are the most commonly used nontrivial data structures, and the most popular implementation on standard hardware uses linear probing, which is both fast and simple."

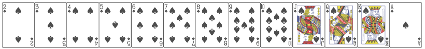


Applications



Most of the problems are trivial or almost trivial after the input sorted.

But the point is, how to solve them in $O(n)$ time, without sorting.

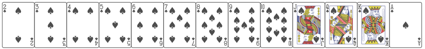


First element occurring k times

$$[1, 2, 3, 2, 1, 4, 5, 8, 6, 7, 4, 2] \rightarrow 1(k = 2), 2(k = 3)$$

Naïve	For each element, count the number of its occurrences, in time $O(n^2)$.
Efficient	Use a hash table to store the counts.

Solution

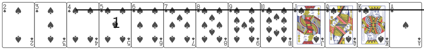


Duplicate elements within k distance from each other[1]

$$[1,2,3,4,1,2,3,4] \rightarrow \texttt{true}(k=4), \texttt{false}(k=3)$$

Naïve	Try each element in order, in time $O(kn)$.
Efficient	Use a hash table to store the index of the previous occurrence of each element.

Solution



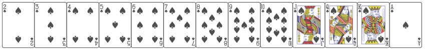
[1] e.g., no worker is supposed to work overtime twice in a week.

Maximum distance between two occurrences of a same element

$$[3, 2, 1, 2, 1, 4, 5, 8, 6, 7, 4, 2] \to 10$$

Naïve	Try each element in order, in time $O(n^2)$.
Efficient	Use a hash table to store the index of the first occurrence of each element.

Solution

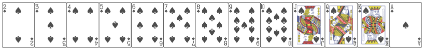


Group elements, ordered by first occurrence

$$[4, 6, 9, 2, 3, 4, 9, 6, 10, 4] \rightarrow [4, 4, 4, 6, 6, 9, 9, 2, 3, 10]$$
$$[5, 3, 5, 1, 3, 3] \rightarrow [5, 5, 3, 3, 3, 1]$$

Naïve	For each element, print all its occurrences, in time $O(n^2)$.
Sort	Sort the array, get the counts for each element. Build an array of counts (e.g., [(1, 1), (3, 3), (5, 2)]). Traverse the input array and do binary search in the array of counts (change the count to 0 after it's done), in $O(n \log n)$.
Efficient	Use a hash table to store the count of each element.

Solution

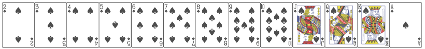


Are two given sets A and B disjoint? $|A| = m, |B| = n$

$$[12, 34, 11, 9, 3], [2, 1, 3, 5] \rightarrow \texttt{false}.$$

Naïve	For each element in A, check whether it's in B, in time $O(mn)$.
Sorting	Sort both arrays and check in order (similar as merging), in $O(n \log n + m \log m)$.
Sorting	Sort one array, and for each element of the other array, do binary search in the sorted one. (Which one should you sort?)
Efficient	Use a hash table to store the elements of the first set.

Solution

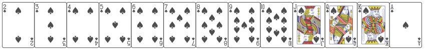


Find symmetric pairs

$$[(11, 20), (30, 40), (5, 10), (40, 30), (10, 5)] \rightarrow (30, 40), (5, 10)$$

Naïve	Try each pair in order, in time $O(n^2)$.
Sort	Sort the pairs by the first coordinate, then search in a smart way, in time $O(n \log n)$.
Efficient	Use a hash table, with x as the key and y the value.

Solution

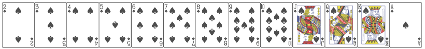


Summary

A data structure that supports insert, search, and delete in average constant time.

The costs:

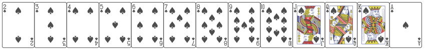
- operations `min`, `max`, `predecessor`, `successor` take $O(n)$ time;
- it cannot be sorted *internally*; and
- performance may degrade significantly when the table is “too full” or “too sparse.”



Dictionary

It’s common to use a dictionary as the motivational example for hash maps.

Shall we use a hash table to implement a dictionary?

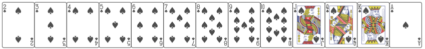


Dictionary

- It's extremely rare that we delete words from a dictionary.
- We don't add new words frequently (Merriam-Webster; Oxford[2]).
- The majority of the operations are searches (updating an item is same as searing from algorithmic perspective).
- Finding the previous and next words are important for dictionaries.
- A sorted array works very well. $\lceil \log 50000 \rceil = 16$
- The performance can be improved by, e.g., keeping record of the positions of the first words starting from a, b…

[2] Most of them are updates instead of new words.

Conclusions

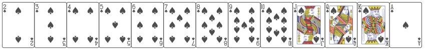


Seven functions

Which operations can be done in the following time?

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$

- $O(n^c), c \geq 3$
- $O(2^n)$ and $O(n!)$

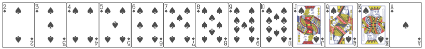


$O(1)$

One cannot do much interesting things in $O(1)$ time.

- `insert` at an unsorted array;
- `push`/`pop`; `enqueue`/`dequeue`;
- `insertFirst`/`removeFirst`;
  and `insertLast` when there exists the `tail` reference;
- `getMax` from a heap;
- all three operations (average-case) in a hash table, when the load factor is low.

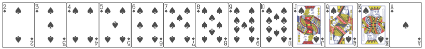
On the other hand, a majority of our algorithms use $O(1)$ space.
Notable exceptions: mergesort and quicksort.



$O(1)$

Which can be done in $O(1)$ time?

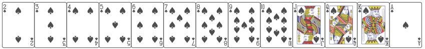
- odd Given an array of n integers, determine whether there exists a pair of numbers in it whose sum is odd.
- even Given an array of n integers, determine whether there exists a pair of numbers in it whose sum is even.



$O(\log n)$

- Since $\log(10^9) \approx 30$, an $O(\log n)$-time algorithm is very good.
- Such an algorithm usually involves a tree structure (sometimes implicitly).

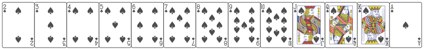
- binary search in a sorted array; similarly, finding a peak.
  Behind it is a balanced binary search tree.
- Most operations in a balanced binary search tree.
- `insert/removeMax` in a binary heap.



$O(n)$

- Any task that requires to read the input takes $\Theta(n)$ time.
- It's thus usually the best one can expect for most nontrivial tasks.
- Fortunately, most tasks in this subject can be done in $O(n)$ time.

- Linear search; `insert/delete` in a sorted array; `delete` in an unsorted array;
- Merge two sorted arrays.
- `deleteLast` in a linked list (whether there is a `tail` reference or not);
- All operations in a binary search tree;
- All three operations (worst-case) in a hash table.

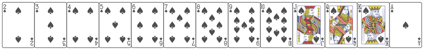


$O(n)$

Let A be an array of size $n \geq 2$ containing integers from 1 to $n-1$.
Design an $O(n)$-time (worst-case) algorithm to find repeated numbers.

- There is exactly one repeated number.
- There is exactly one repeated number and it repeats three times.
- There is exactly two repeated numbers.

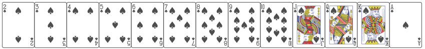
Can you do them with only $O(1)$ extra space?



$O(n \log n)$

- Significantly better than n^2, and slightly worse than n.
- divide and conquer, into $\log n$ levels;
- a data structure having operations $\log n$

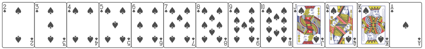
"Good" sorting algorithms.



$O(n^2)$

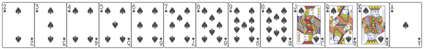
- "Bad" sorting algorithms.
- Most commonly shown as two nested for-loops. But careful!

True or false:
An algorithm on two nested for-loops, both taking n iterations in the worst case, takes $\Theta(n^2)$ time.



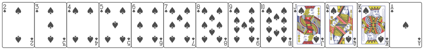
$O(n^c)$ for $c \geq 3$

Directly multiple two $n \times n$ matrices.



$O(2^n)$ and $O(n!)$

- We have not really introduces any algorithm taking time $O(2^n)$ or $O(n!)$.
- But it's very natural: If we need to try every subset of a set of n elements.
- Similarly, if we need to try every permutation of a set of n elements, then we need $O(n!)$.



Not Covered

Uncovered

- operations on (super) large integers ();
- Iterators ();
- error/exception handling ();
- nontrivial analysis techniques ().

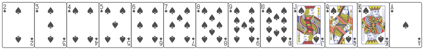
- Problems that probably cannot be solved in polynomial time ();
- Incomputable problems ().

- external-memory algorithms: data that cannot fit into the memory ()

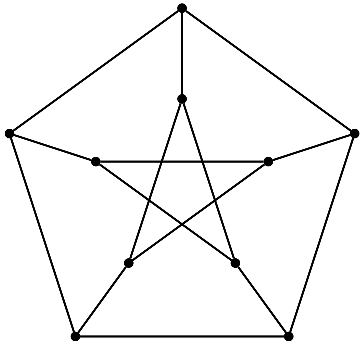
How did Amazon transfer data

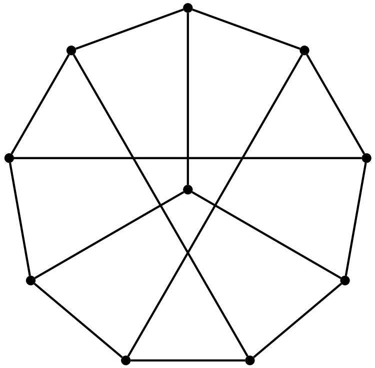
Differences between human beings and computers

	human	computers
trivial calculations	$1/5$ easy	$99776 * 67799$ easy
trivial calculations	$a\%n \geq 0$	`a % n` may < 0
trivial calculations	$\lvert x \rvert \geq 0$	`Math.abs()` may < 0
expressions	$3 * (7 + 2)$	$372 + *$
quantity	$4, 1, 3, 2$	
	the fewer, the easier	border cases are the hardest.
. . . . . .		



Graphs

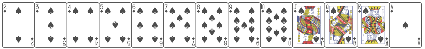




Representation of graphs

$$\begin{array}{ccccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}$$

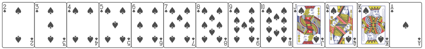
$$\begin{tikzcd}
u_1 & \boxed{} \arrow[r] & \boxed{u_2} \arrow[r] & \boxed{u_4} & \boxed{\perp} \\
u_2 & \boxed{} \arrow[r] & \boxed{u_1} \arrow[r] & \boxed{u_3} & \boxed{\perp} \\
u_3 & \boxed{} \arrow[r] & \boxed{u_2} \arrow[r] & \boxed{u_4} & \boxed{\perp} \\
u_4 & \boxed{} \arrow[r] & \boxed{u_1} \arrow[r] & \boxed{u_3} \arrow[r] & \boxed{v_4} \arrow[r] & \boxed{w_4} & \boxed{\perp} \\
v_1 & \boxed{} \arrow[r] & \boxed{v_2} \arrow[r] & \boxed{v_4} & \boxed{\perp} \\
v_2 & \boxed{} \arrow[r] & \boxed{v_1} \arrow[r] & \boxed{v_3} & \boxed{\perp} \\
v_3 & \boxed{} \arrow[r] & \boxed{v_2} \arrow[r] & \boxed{v_4} & \boxed{\perp} \\
v_4 & \boxed{} \arrow[r] & \boxed{v_1} \arrow[r] & \boxed{v_3} \arrow[r] & \boxed{u_4} \arrow[r] & \boxed{w_4} & \boxed{\perp} \\
w_1 & \boxed{} \arrow[r] & \boxed{w_2} \arrow[r] & \boxed{w_4} & \boxed{\perp} \\
w_2 & \boxed{} \arrow[r] & \boxed{w_1} \arrow[r] & \boxed{w_3} & \boxed{\perp} \\
w_3 & \boxed{} \arrow[r] & \boxed{w_2} \arrow[r] & \boxed{w_4} & \boxed{\perp} \\
w_4 & \boxed{} \arrow[r] & \boxed{w_1} \arrow[r] & \boxed{w_3} \arrow[r] & \boxed{u_4} \arrow[r] & \boxed{v_4} & \boxed{\perp}
\end{tikzcd}$$



Final word

Perhaps the most important principle for the good algorithm designer is to refuse to be content.

Aho, Hopcroft, and Ullman, *the design and analysis of algorithms* (1974)



December 12: Final Exam