

Week 5: Wednesday

EECS 281

Agenda

Dictionary ADT

Hash tables

Hash functions

Collision resolution

Separate chaining

Open addressing: linear probing, quadratic probing, double Hashing

Dynamic hashing

Associative and unordered containers in STL

Cryptographic hashing

Code design

4:45 PM after section

3-way sort

Implement **threeWaySort**, a function that sorts a vector of integers whose values are **0**, **1** and **2**.

Time complexity should be $\Theta(n)$ and space complexity should be $\Theta(1)$.

3-way sort

```
void threeWaySort(vector<int> &v) {  
    size_t countZeros = 0;  
    size_t countOnes = 0;  
    for (auto i : v) {  
        if (i == 0) {  
            countZeros += 1;  
        } else if (i == 1) {  
            countOnes += 1;  
        }  
    }  
    for (size_t i = 0; i < countZeros; i += 1) {  
        v[i] = 0;  
    }  
    for (size_t i = countZeros; i < countZeros + countOnes; i += 1) {  
        v[i] = 1;  
    }  
    for (size_t i = countZeros + countOnes; i < v.size(); i += 1) {  
        v[i] = 2;  
    }  
}
```

Midterm Exam

Common problems

Comparing an iterator to nullptr

Dereferencing iterator and pointers before checking its validity

Using operators or functions not defined for forward iterators

Variable name misspellings

Use ++ on a pointer in a linked list

Dereferencing a pointer (to a struct) to get an int







People You May Know



Reggie Campus Corgi

18 mutual friends including Ashirvad Varma and Carolyn Zhou



Add Friend



Follow



Message



More

I'm Reggie the Campus Corgi. I'm a trained service dog and love making new friends on campus.



Studies Public Relations at **Univeristy of Michigan**



Lives in **Ann Arbor, Michigan**



From **Ann Arbor, Michigan**

✕ People You May Know 🔍



Reggie Campus Corgi

18 mutual friends including Ashirvad Varma and Carolyn Zhou

-  Add Friend
-  Follow
-  Message
-  More

I'm Reggie the Campus Corgi. I'm a trained service dog and love making new friends on campus.

🎓 Studies Public Relations at **Univeristy of Michigan**

🏠 Lives in **Ann Arbor, Michigan**

📍 From **Ann Arbor, Michigan**



Reggie Campus Corgi with Maxim Aleksa.

20 May at 14:54 · 🌐 ▼

My new friend!



👍 Like 💬 Comment ➦ Share

👍❤️🐶 Linda Sun, Kevin Lee and 176 others



People You May Know



Reggie Campus Corgi

18 mutual friends including Ashirvad Varma and Carolyn Zhou



Add Friend



Follow



Message



More

I'm Reggie the Campus Corgi. I'm a trained service dog and love making new friends on campus.



Studies Public Relations at **Univeristy of Michigan**



Lives in **Ann Arbor, Michigan**



From **Ann Arbor, Michigan**



🔍 Reggie Campus Corgi



Reggie Campus Corgi with Maxim Aleksa. ▾

20 de maig at 14:55 · 🌐

My new friend!



Linda Sun and 178 others

11 Comments







Someone in 2010 bought 2 pizzas with 10,000 bitcoins – which today would be worth \$20 million



Rob Price

May 22, 2017, 6:11 AM 77,090



FACEBOOK



LINKEDIN



TWITTER



On May 22, 2010, a developer bought two pizzas using 10,000 units of a then-little-known digital currency called bitcoin.

Today, 10,000 bitcoins are worth more than \$20 million (£15.4 million).

Bitcoin is going nuclear. Its price is tearing upward, with each bitcoin worth \$2,128 (£1,638) — a little shy of its all-time-high of \$2,185 (£1,682) reached earlier Monday morning.

Just a year ago, it was trading at just \$443 (£341), after deflating from what was then seen as the giddy highs of about \$1,100 (£847) in late 2013. It has since embarked on an epic bull run.



Laszlo Hanyecz bought these pizzas for 10,000 bitcoins on May 22, 2010. Laszlo Hanyecz

Dictionary ADT

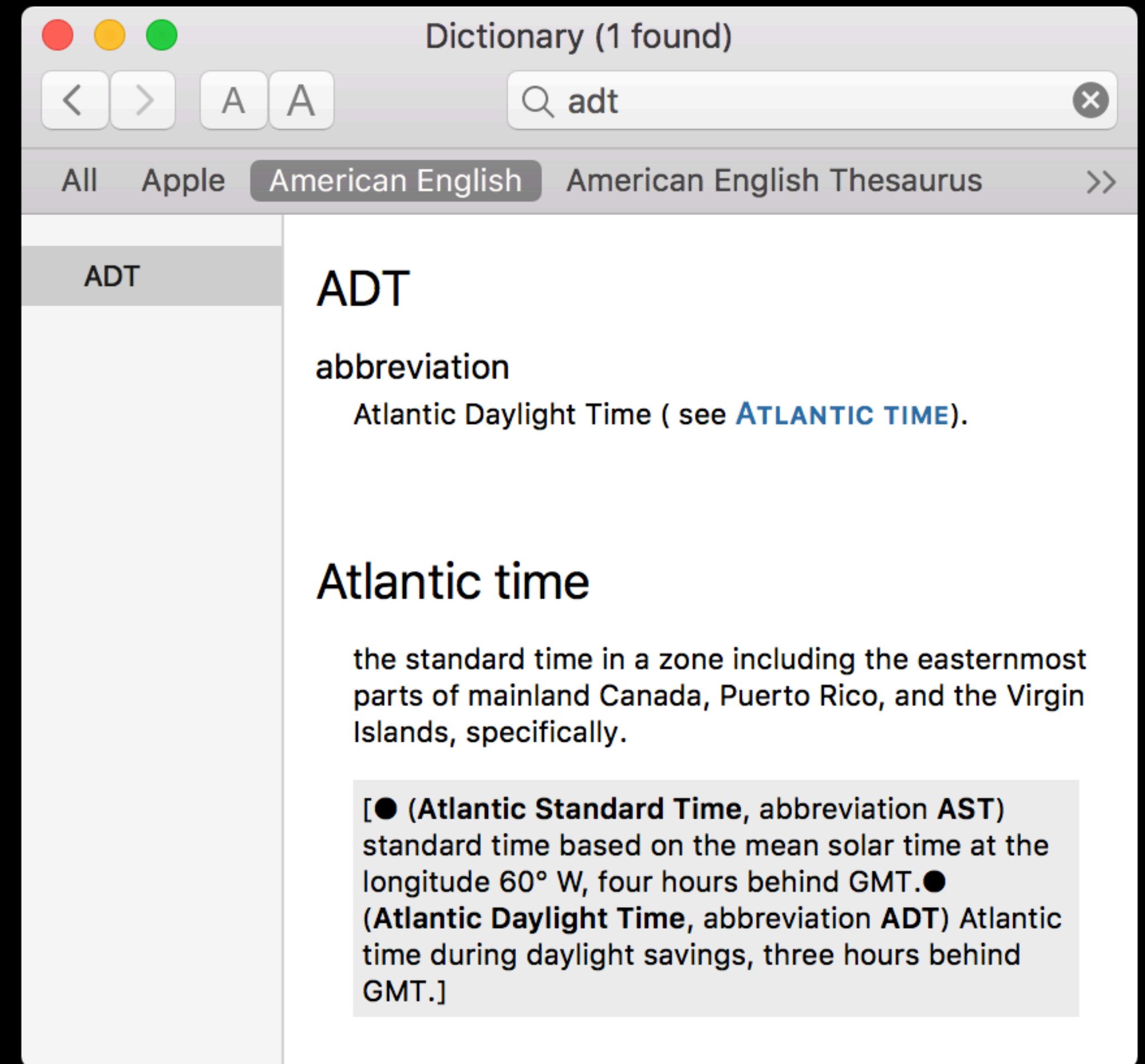
Dictionary

Abstract data type

`insert(key, value)`

`find(key)`

`remove(key)`



Dictionary

Airport code → Airport

"DTW"	"Detroit"
"LHR"	"London Heathrow"
"SFO"	"San Francisco"
"YYZ"	"Toronto Pearson"

Array with classes

What's wrong with this?

```
class Airport {  
public:  
    string code;  
    string name;  
};  
  
Airport airports[] = // ...
```


Hash table

Data structure

Hash table

Abstract data type

Dictionary

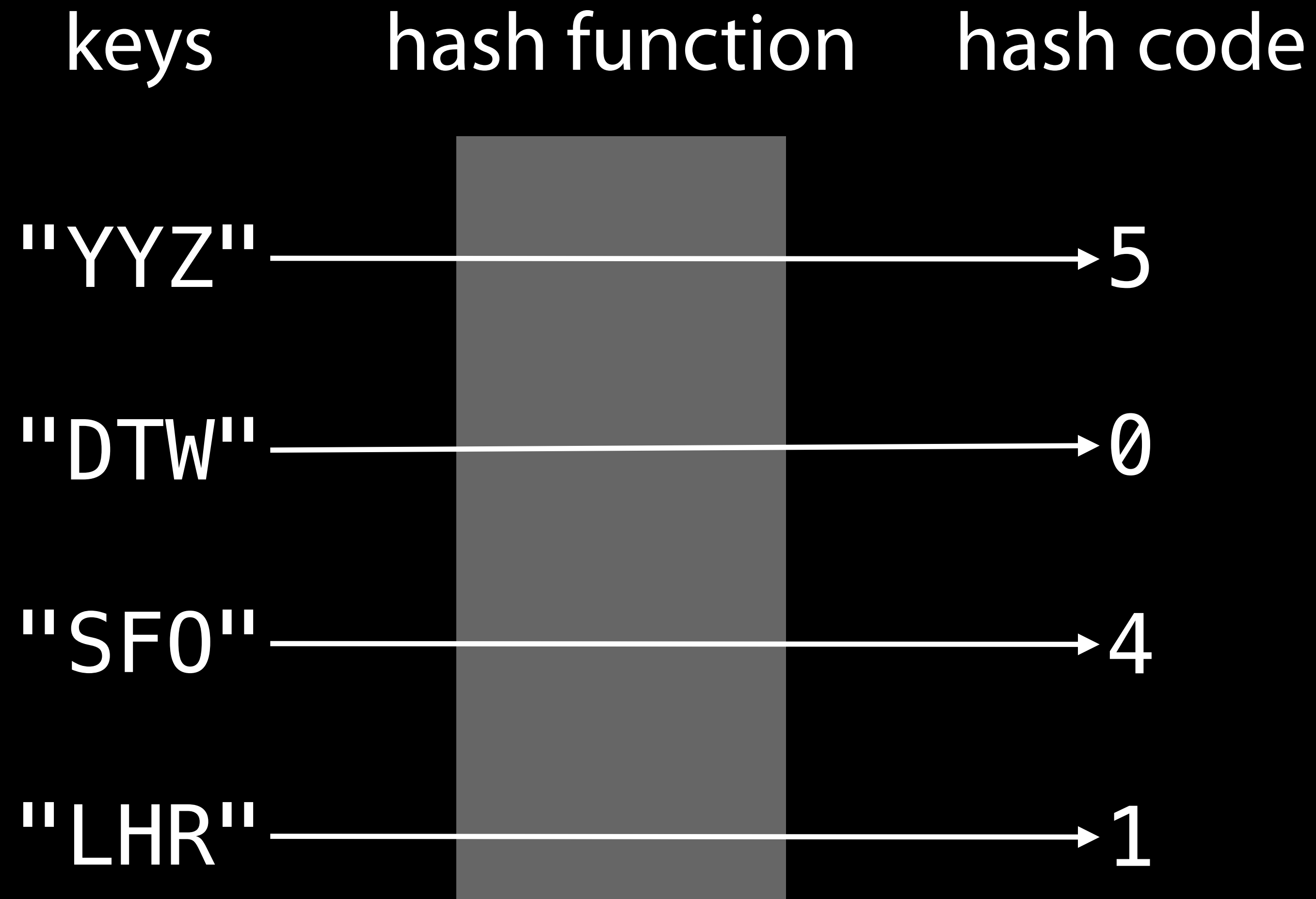
Associative array

Map

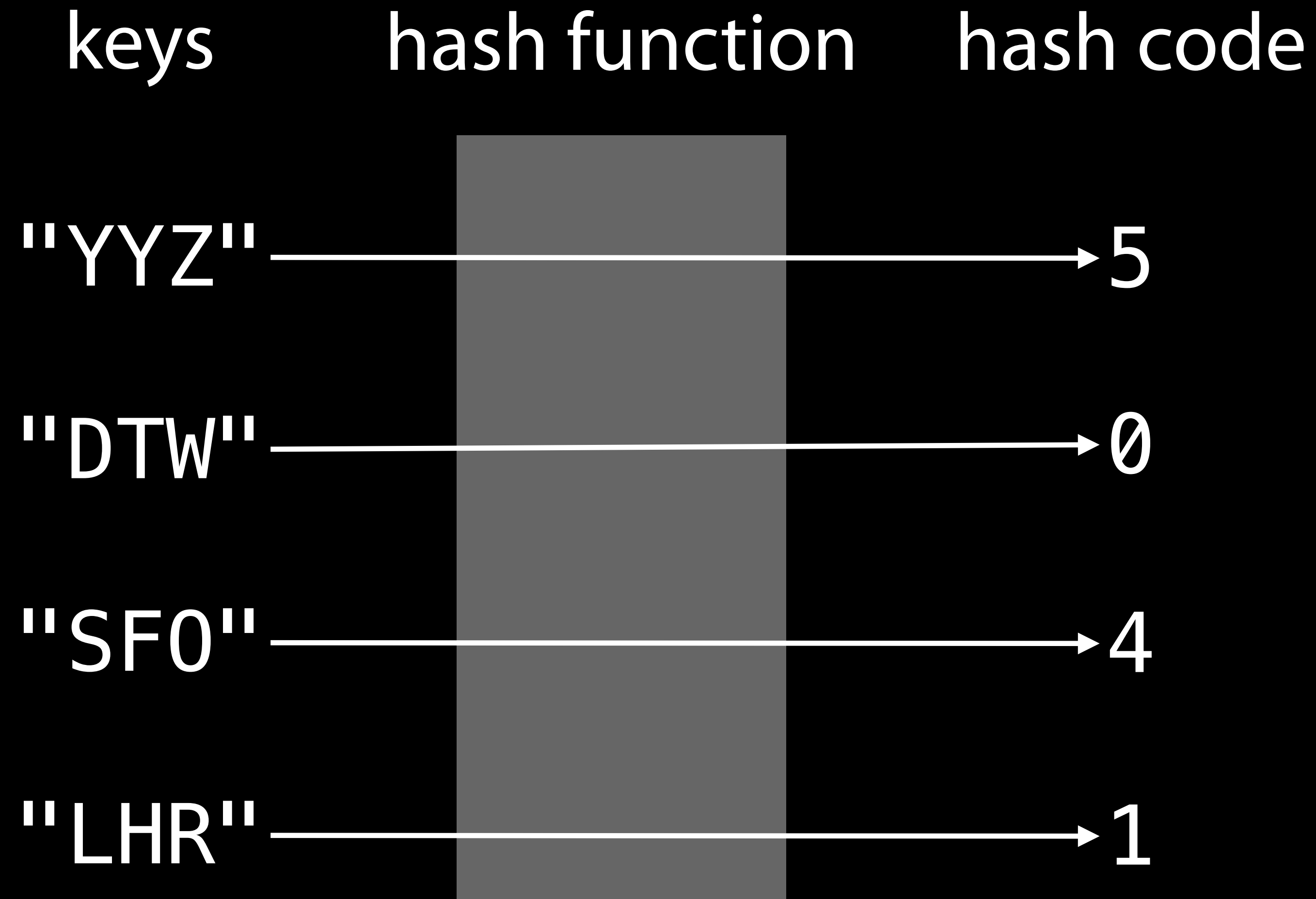
Symbol table

(key, value) pairs

Hash table



Hash table



0	"Detroit"
1	"London Heathrow"
2	
3	
4	"San Francisco"
5	"Toronto Pearson"
6	
7	

Pseudocode

```
insert(key, value):  
    index = hashFn(key)  
    array[index] = (key, value)
```

```
find(key):  
    index = hashFn(key)  
    return array[index]
```

```
remove(key):  
    index = hashFn(key)  
    array[index] = null
```

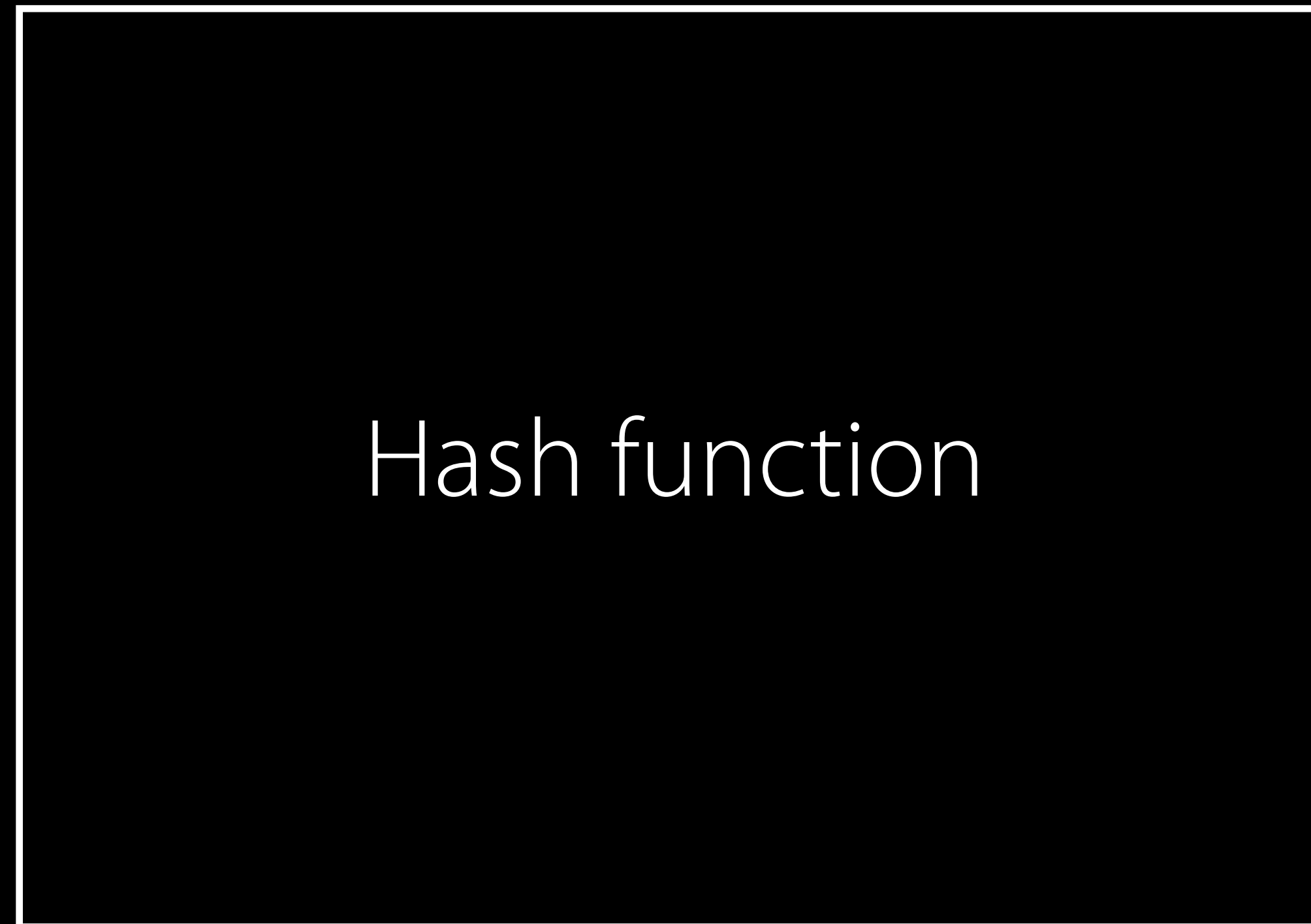
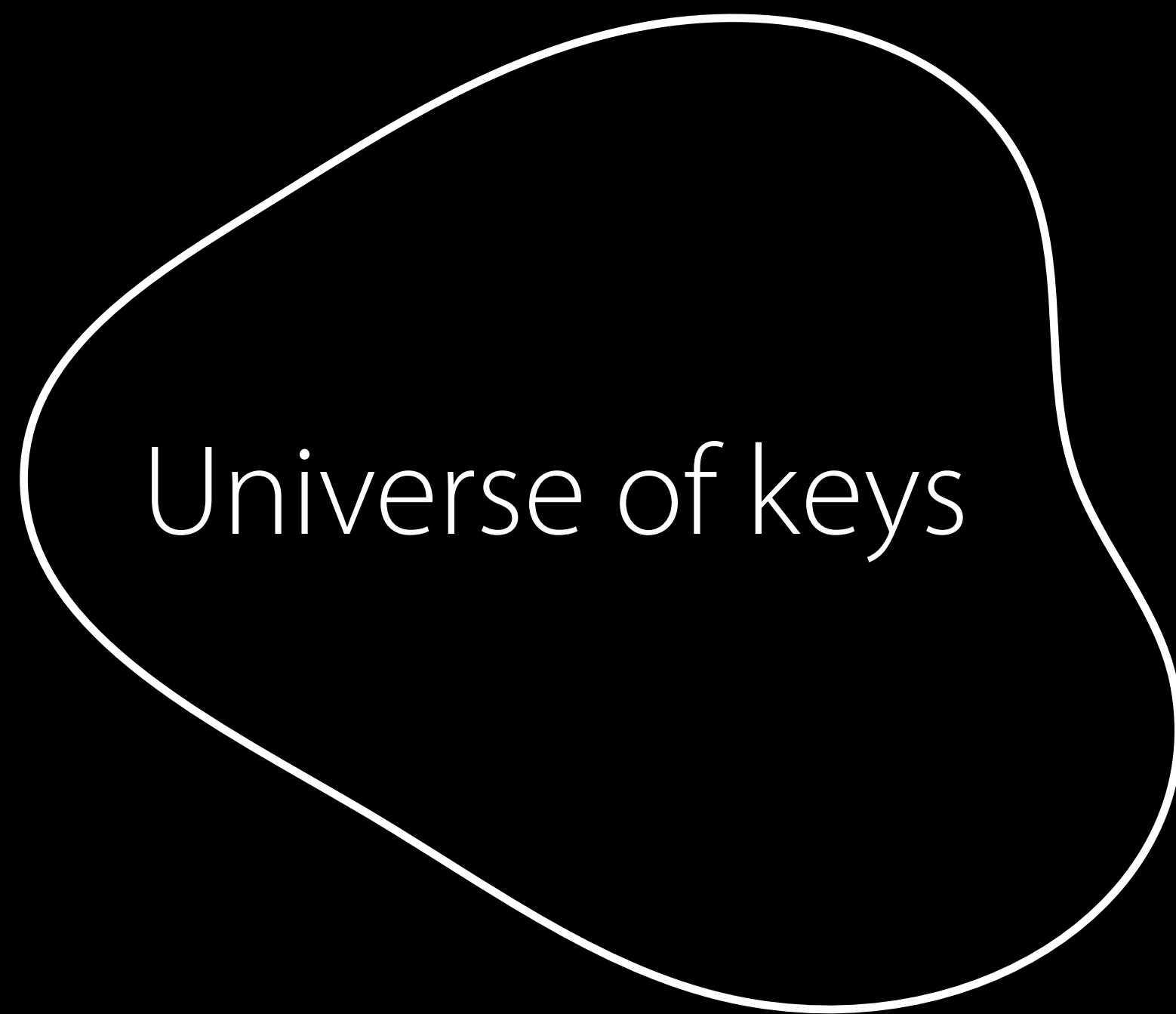

Hash table performance

Search / insert / delete

Average: $O(1)$

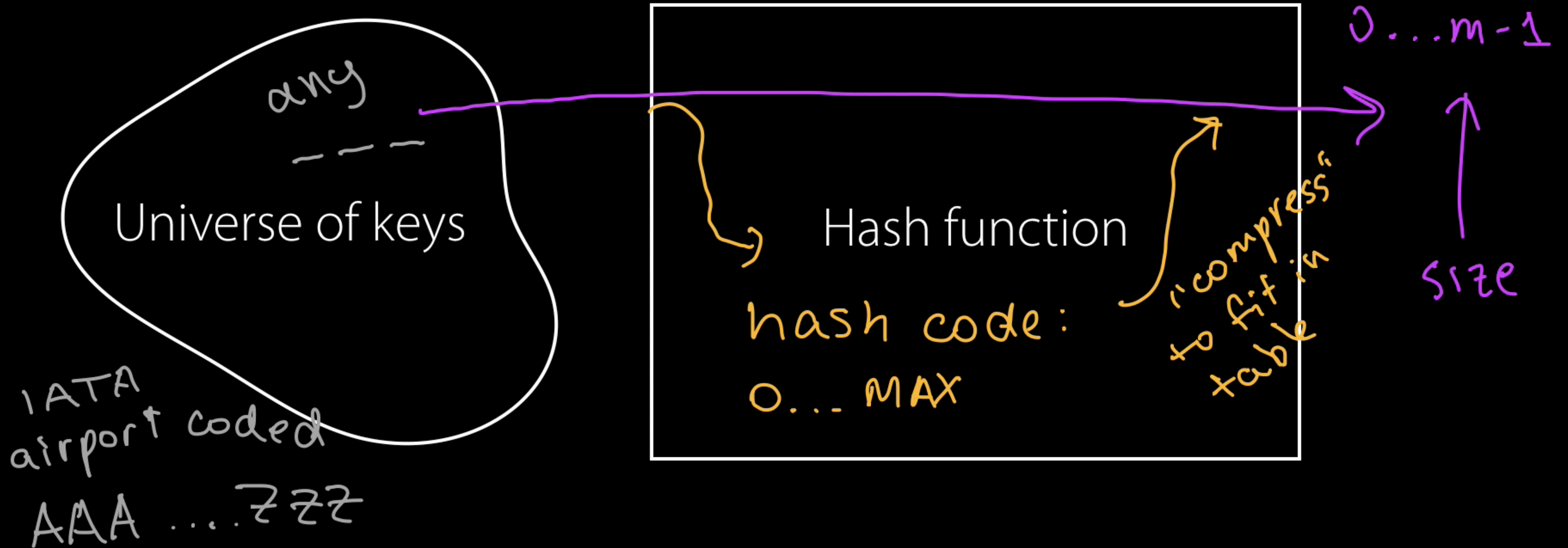
Worst case: $O(N)$, due to collisions

Hash functions



Hash functions

n : #elt stored
 m : size of table/array
 $|U|$: size of universe



Hash functions

What are the hash codes of "DTW", "LHR", "SFO" and "YYZ"?

```
int airportHash(string airportCode) {  
    if (airportCode.length() != 3) {  
        return -1;  
    }  
    else {  
        return toupper(airportCode[0]) - 'A';  
    }  
}
```


Hash functions

What are the hash codes of "DTW", "LHR", "SFO" and "YYZ"?

```
int airportHash(string airportCode) {  
    if (airportCode.length() != 3) {  
        return -1;  
    }  
    else {  
        return toupper(airportCode[0]) - 'A';  
    }  
}
```

diff between first letter & A
Collisions SYD SFO

Hash functions

What is a good hash function for an airport code?

Hash functions

What is a good hash function for a ~~string~~? *airport code?*
want hash code *e.g. "DTW"*

031922

wasting space

↑ ↑ ↑
3 19 22

$\text{first \#} \times 26^2 + \text{second \#} \times 26^1 + \text{third \#}$

$\rightarrow 25441$

then

mod by m (size)

Hash functions

What is a good hash function for a string?

Hash functions

What is a good hash function for a string? S

$$\text{hashCode} = s[0] \times \underline{31}^{n-1} + s[1] \times \underline{31}^{n-2} + \dots + s[n-1] \times 31^0$$

(ASCII)

① Mult. by 31 $\Leftrightarrow$ mult. by 32 - 1
 $\ll 5$
0101 $\Rightarrow$ 010100000

unordered_map < Person, Color >
teach C++ how to hash

Compression

Simple compression function:

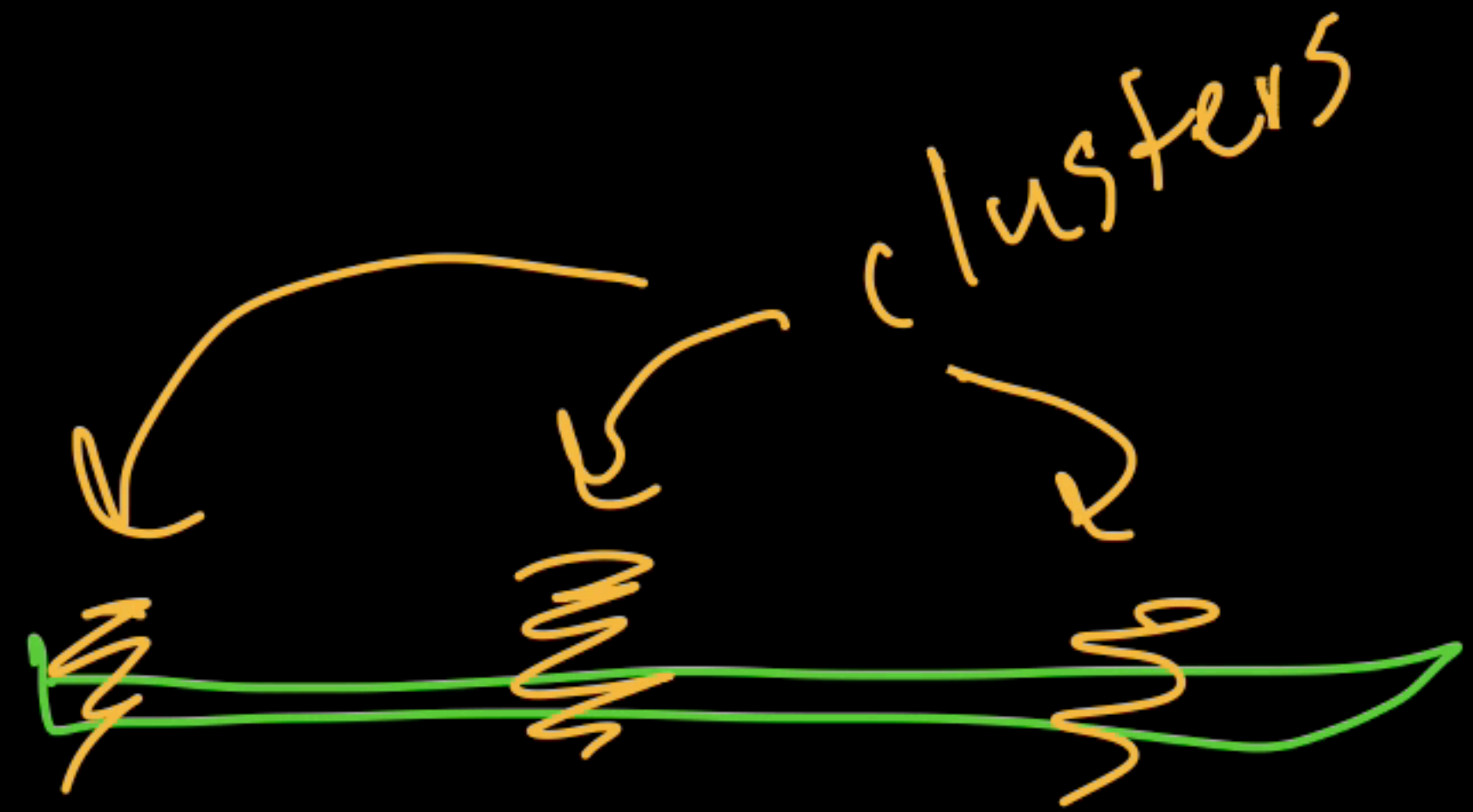
$$h(i) = i \% SIZE$$

Compression

Simple compression function:

$$h(i) = i \% \text{SIZE}$$

common factors → 2?



improve by making
table size prime

Compression

Simple compression function:

$$h(i) = i \% SIZE$$

Better compression function:

$$h(i) = ((a * i + b) \bmod p) \bmod SIZE$$

Compression

Simple compression function:

$$h(i) = i \% \text{SIZE}$$

(also m)
size of array
used in hash
table

Better compression function:

$$h(i) = ((a * i + b) \bmod p) \bmod \text{SIZE} \quad \geq \# \text{ of elems stored}$$

$\Theta(n)$ or $\Omega(n)$

$\leftarrow$ v large prime int
 $p \gg \text{SIZE}$

$\leftarrow$ any hash code

$\nearrow$
randomly
chosen ints
at beginning

Good hash functions

Uniformly distributed output across the size of the hash table.

Very different hash indices for inputs that only differ slightly.

Very simple, quick operations such as bitwise manipulations.

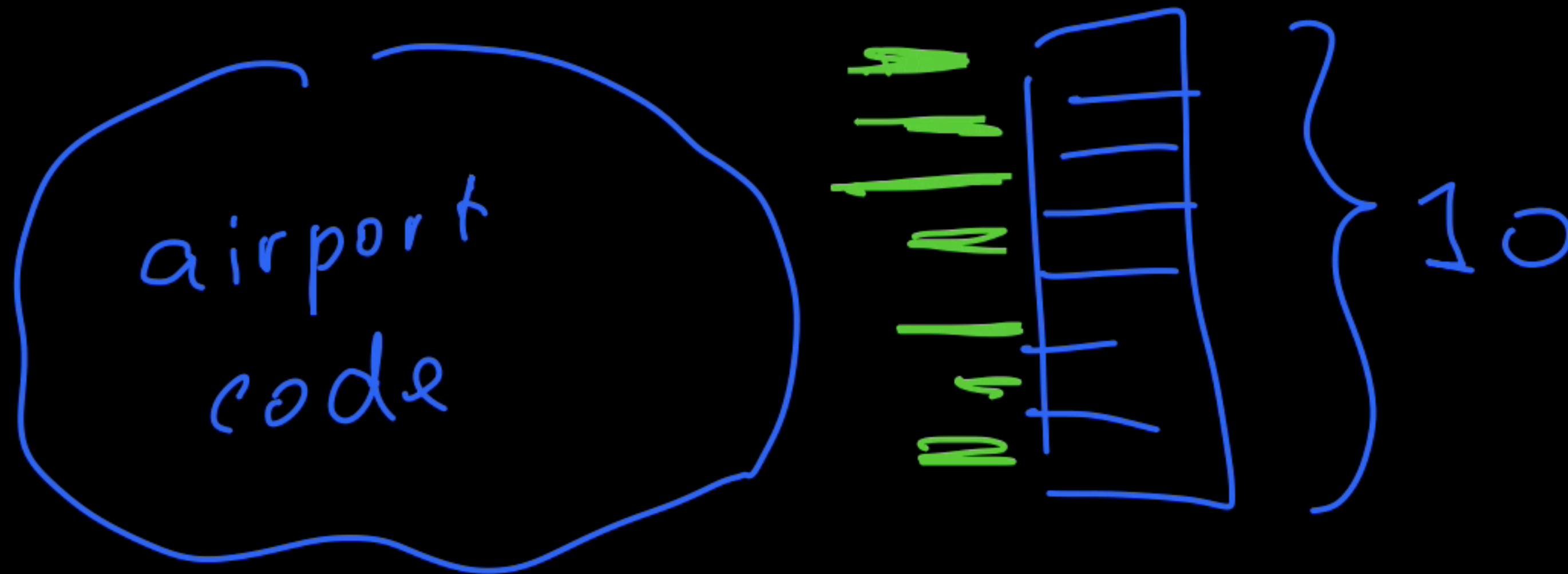
Good hash functions

no clusters

Uniformly distributed output across the size of the hash table.

Very different hash indices for inputs that only differ slightly.

Very simple, quick operations such as bitwise manipulations.



Collision resolution

Collision: two keys hash to the same index in the array representing the hash table

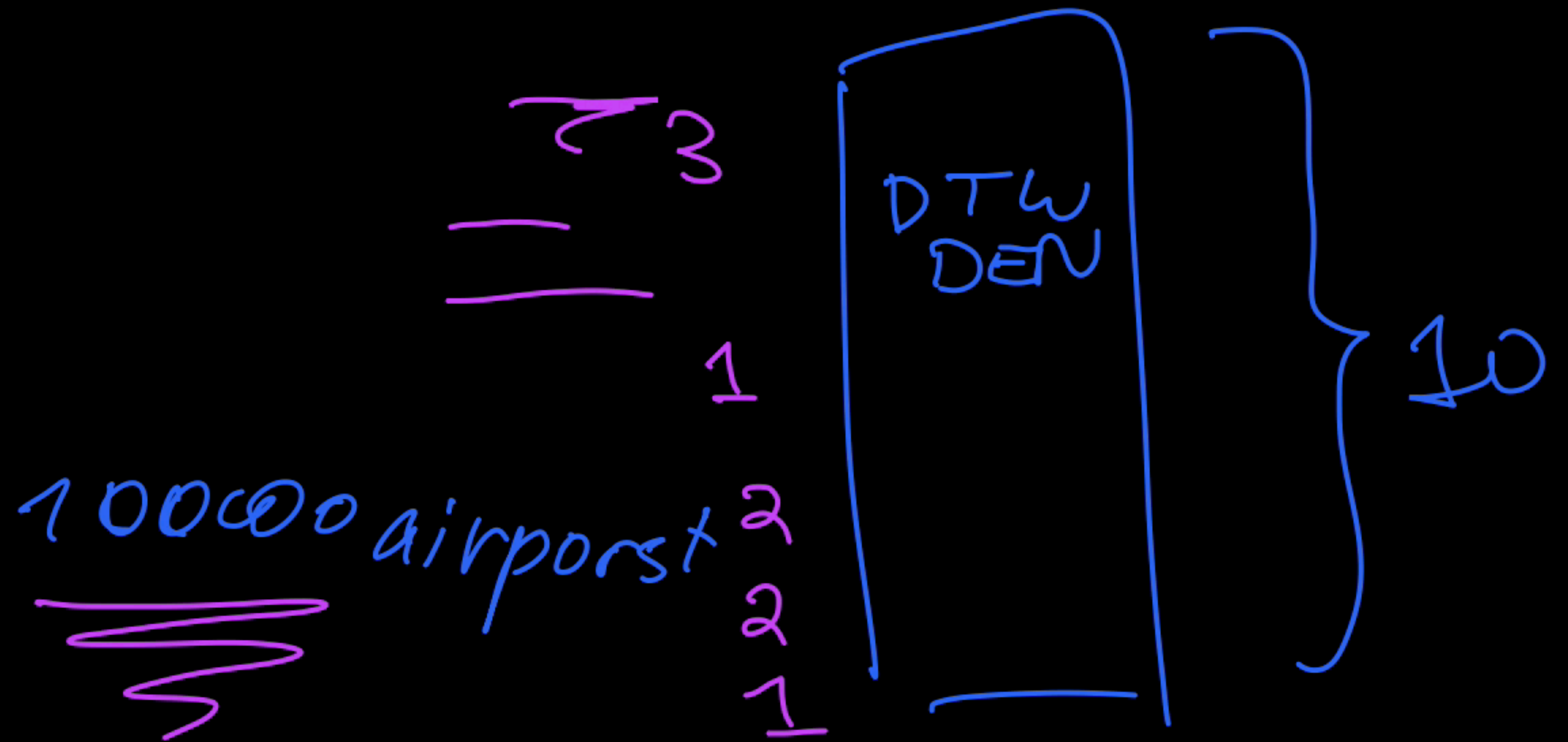
Separate chaining

Open addressing

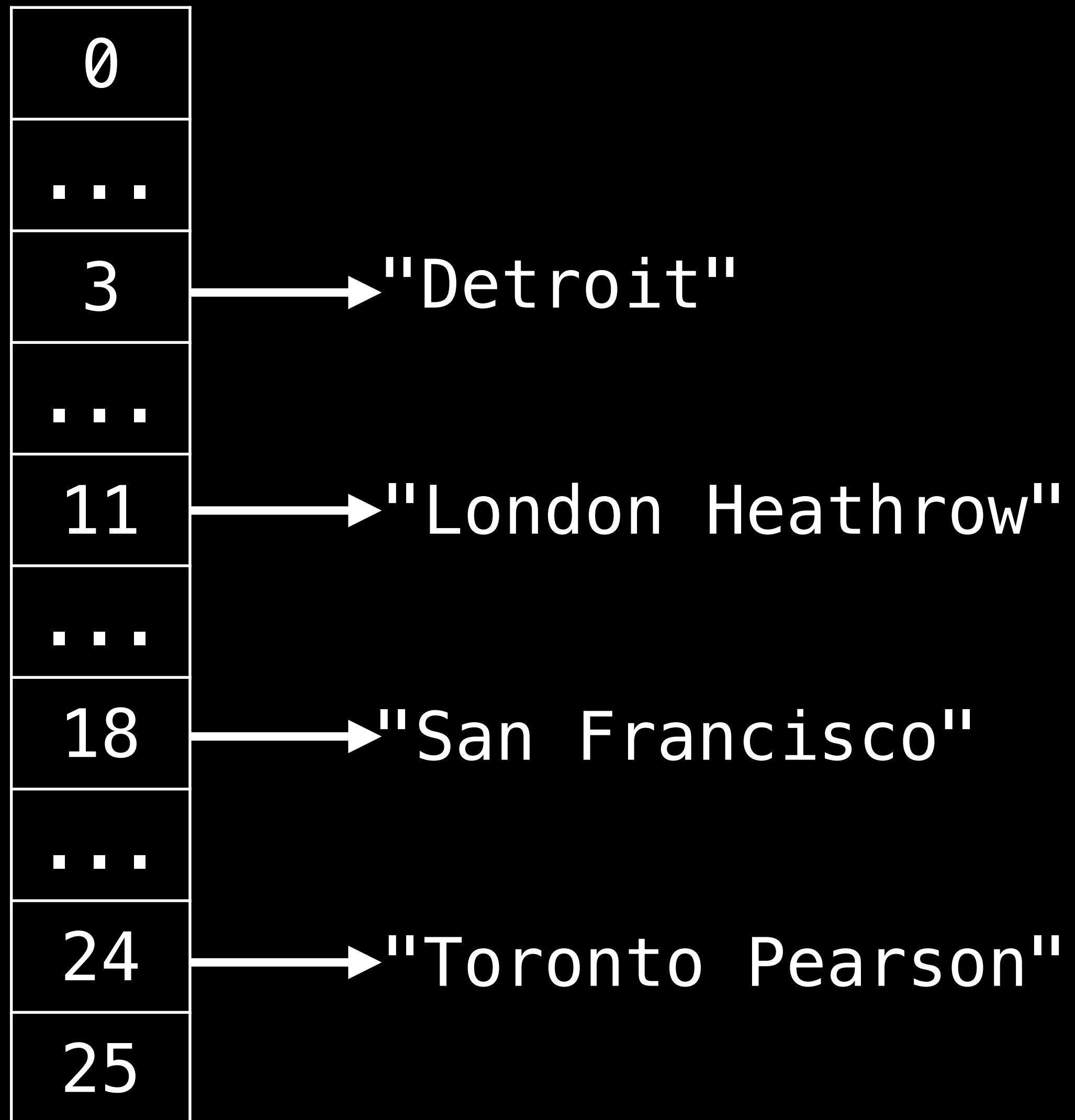
Collision resolution

Collision: two keys hash to the same index in the array representing the hash table

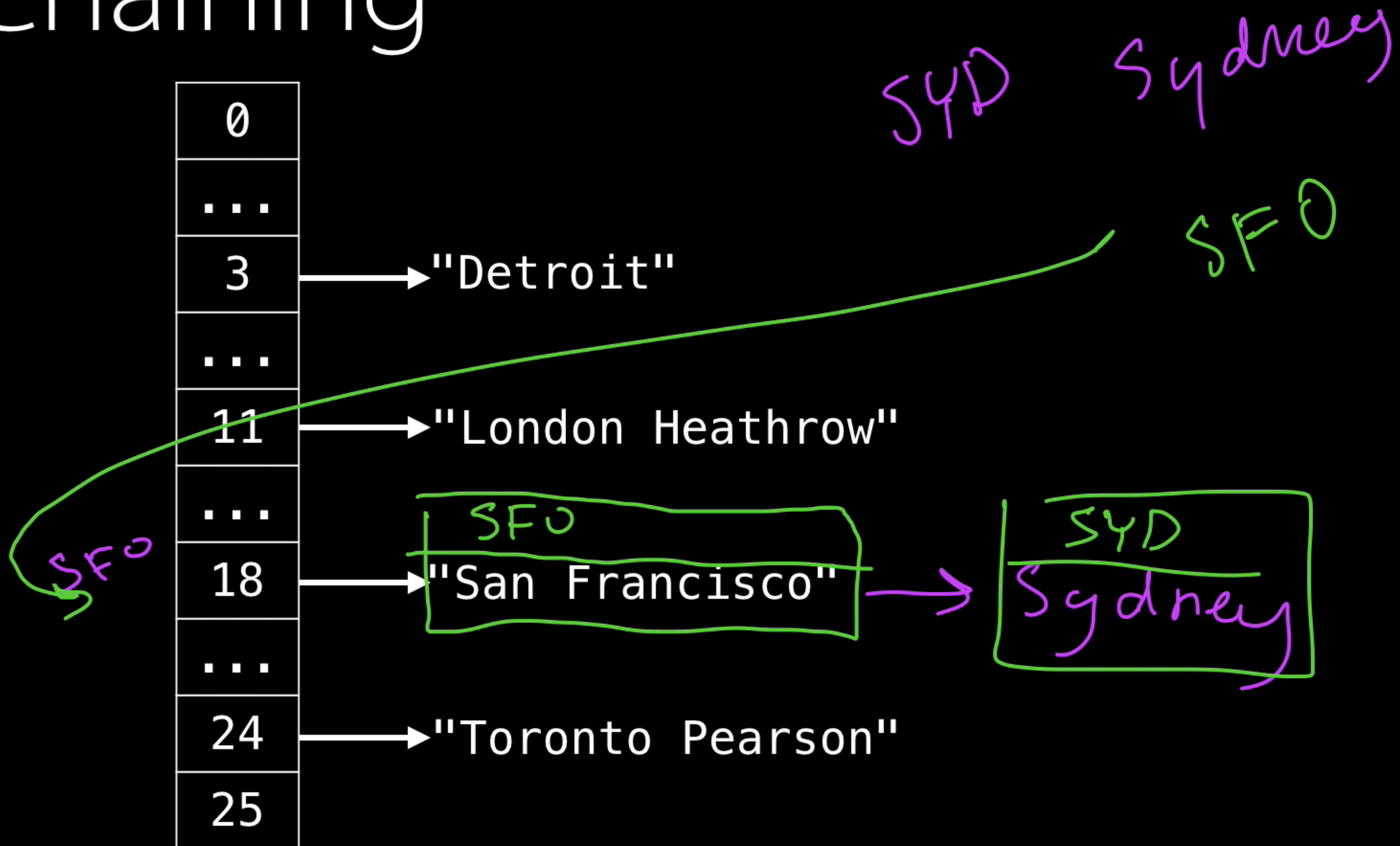
- ① Separate chaining
- ② Open addressing



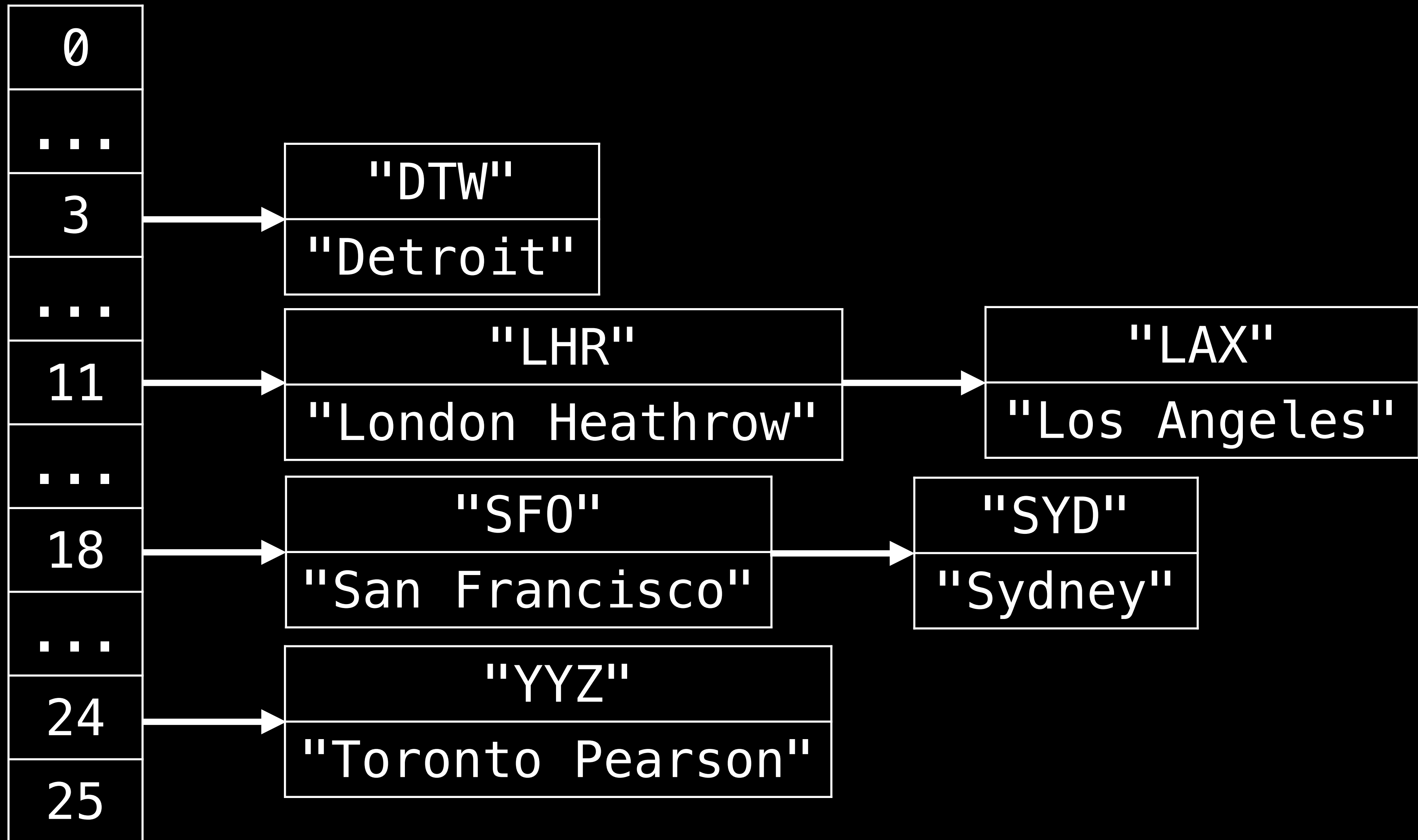
Chaining



Chaining



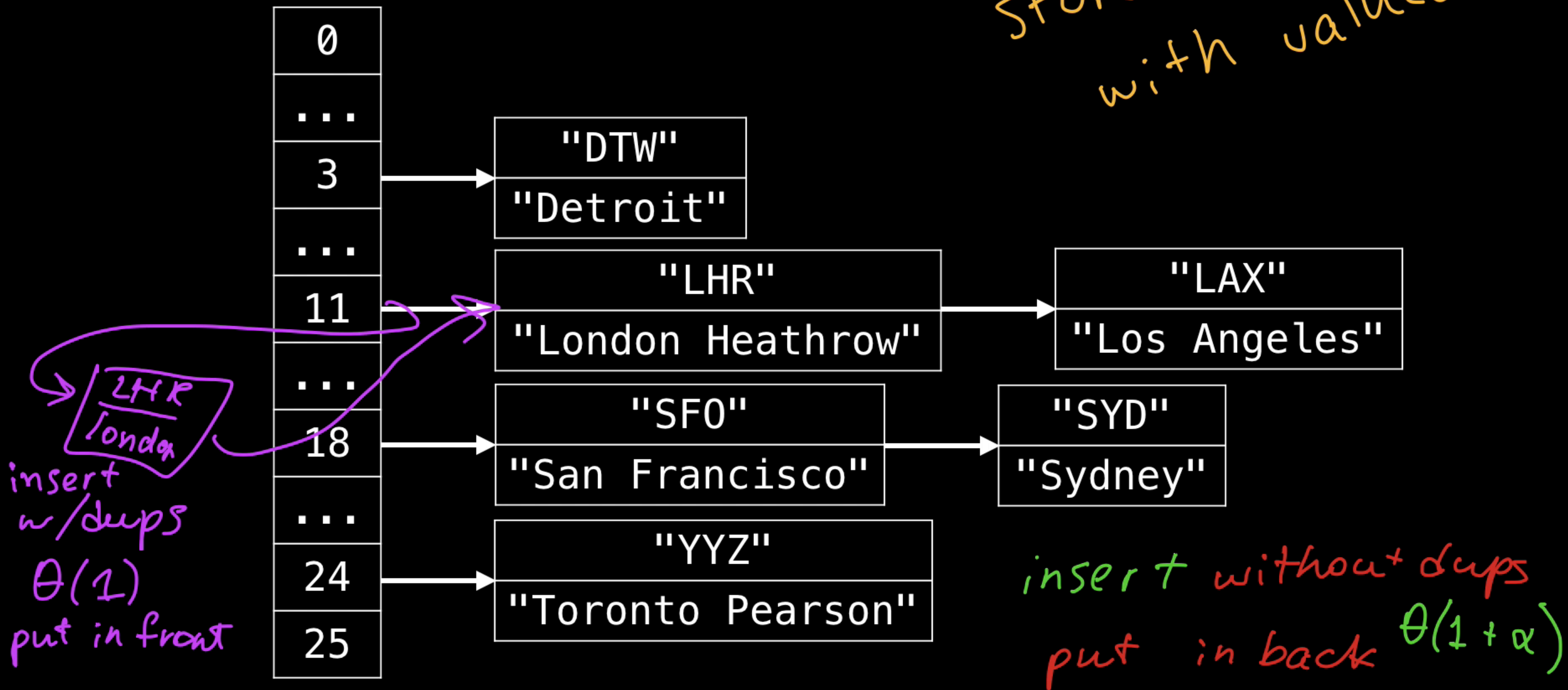
Chaining



Chaining

"LHR" : "London"

store keys
with values



Open addressing

Open addressing

each entry: store 1 key-val pair

if collision $\rightarrow$ find another index

Linear probing

Open addressing

Linear probing $h(k, i) = h'(k) + i$

Quadratic probing $h(k, i) = h'(k) + i^2$

Double Hashing $h(k, i) = h_1(k) + i * h_2(k)$

Open addressing

until empty/null

size of array
 $\geq$ # of elems
stored

Linear probing

$$h(k, i) = h'(k) + i$$

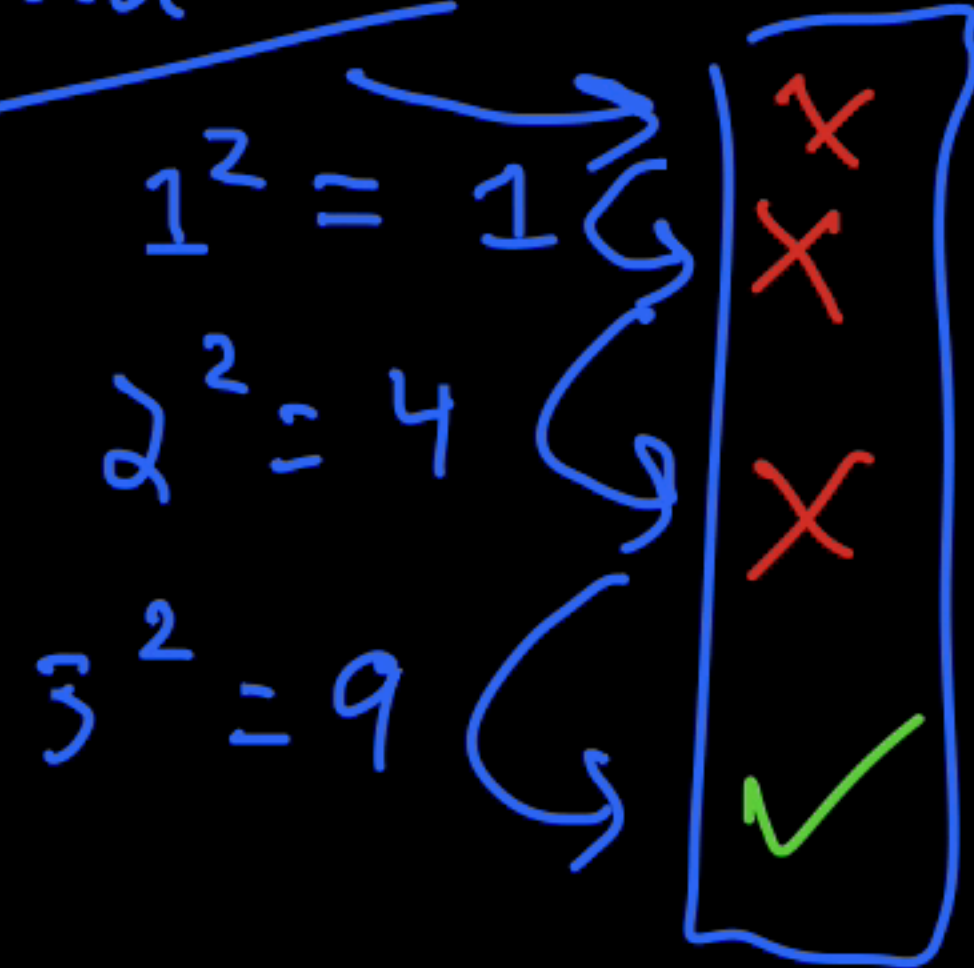
Quadratic probing

$$h(k, i) = h'(k) + i^2$$

Double Hashing

$$h(k, i) = h_1(k) + i * h_2(k)$$

Quadratic



tries to
solve
clustering

of failed
attempts

Double Hashing



Linear probing

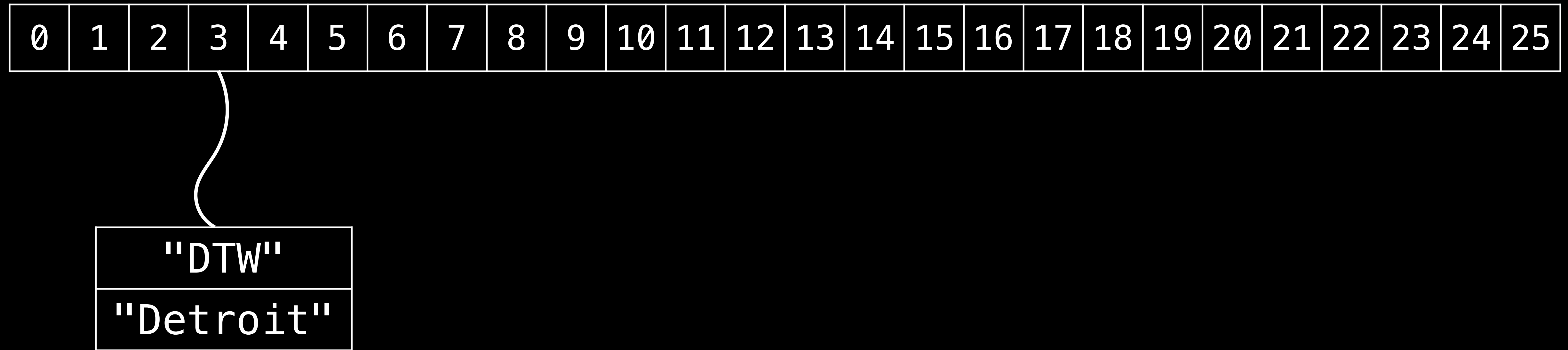
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Linear probing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

"DTW"
"Detroit"

Linear probing



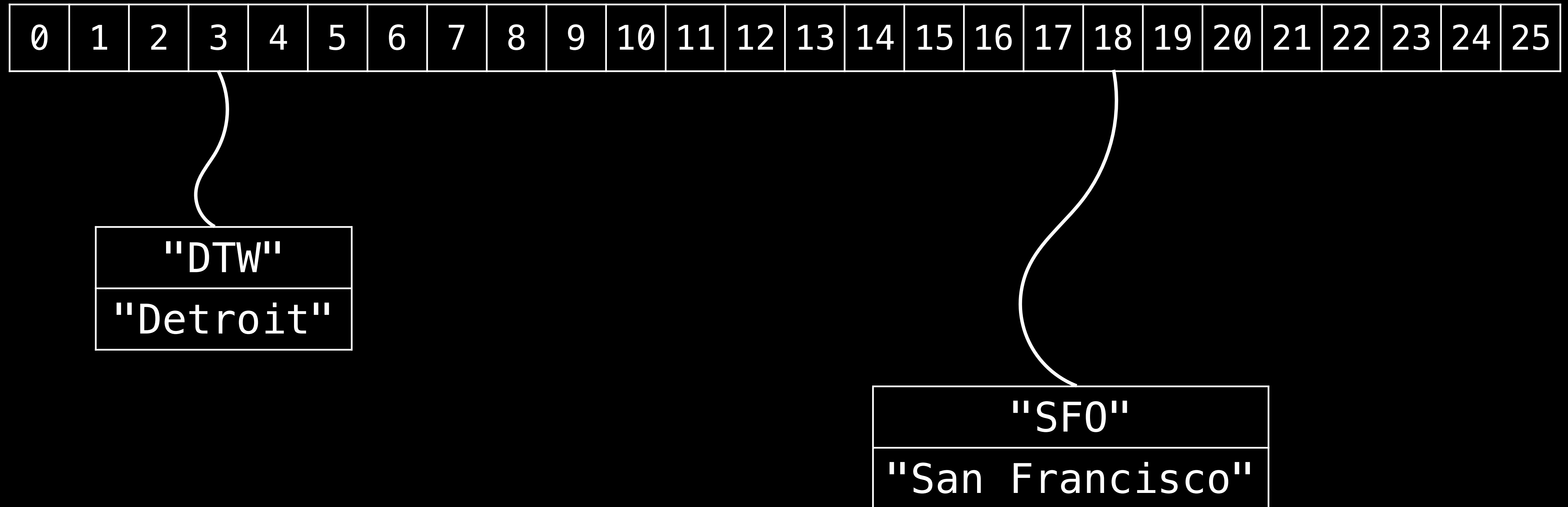
Linear probing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

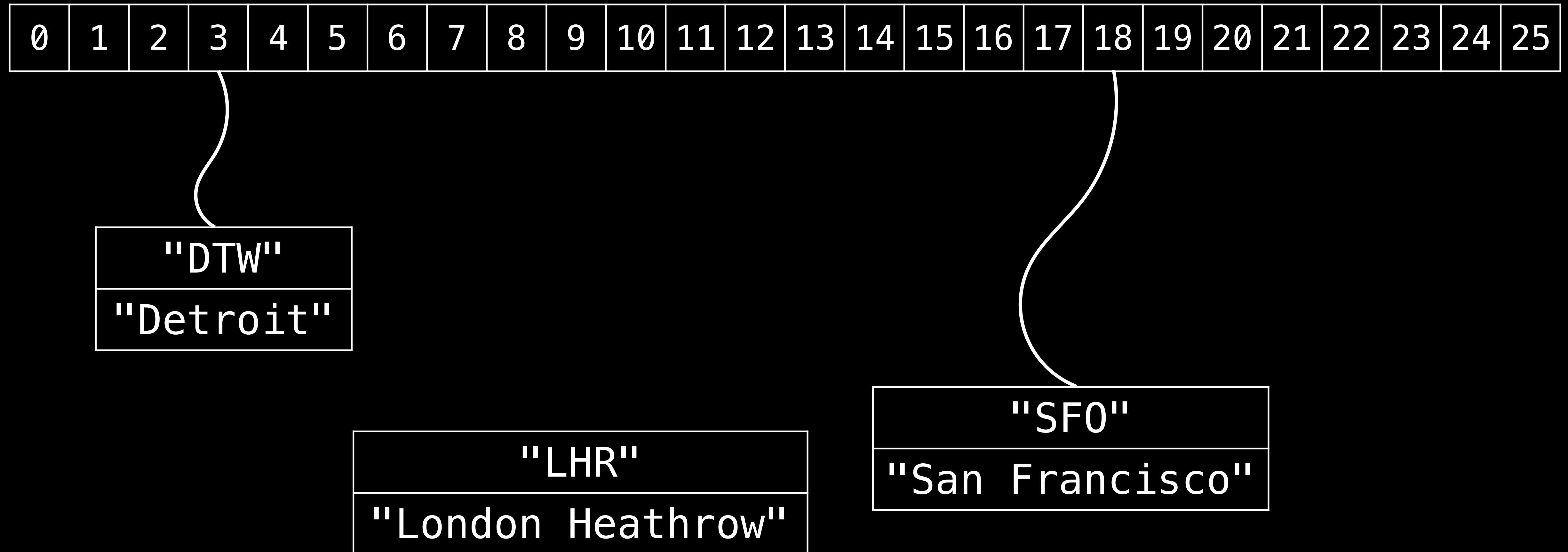
"DTW"
"Detroit"

"SF0"
"San Francisco"

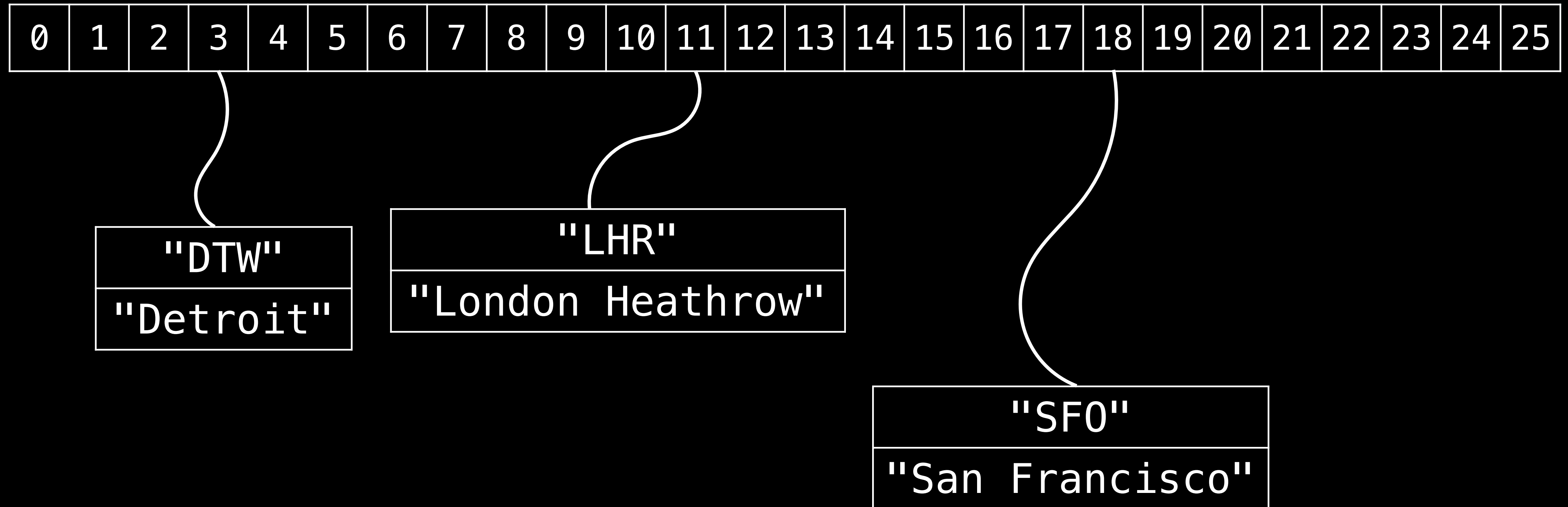
Linear probing



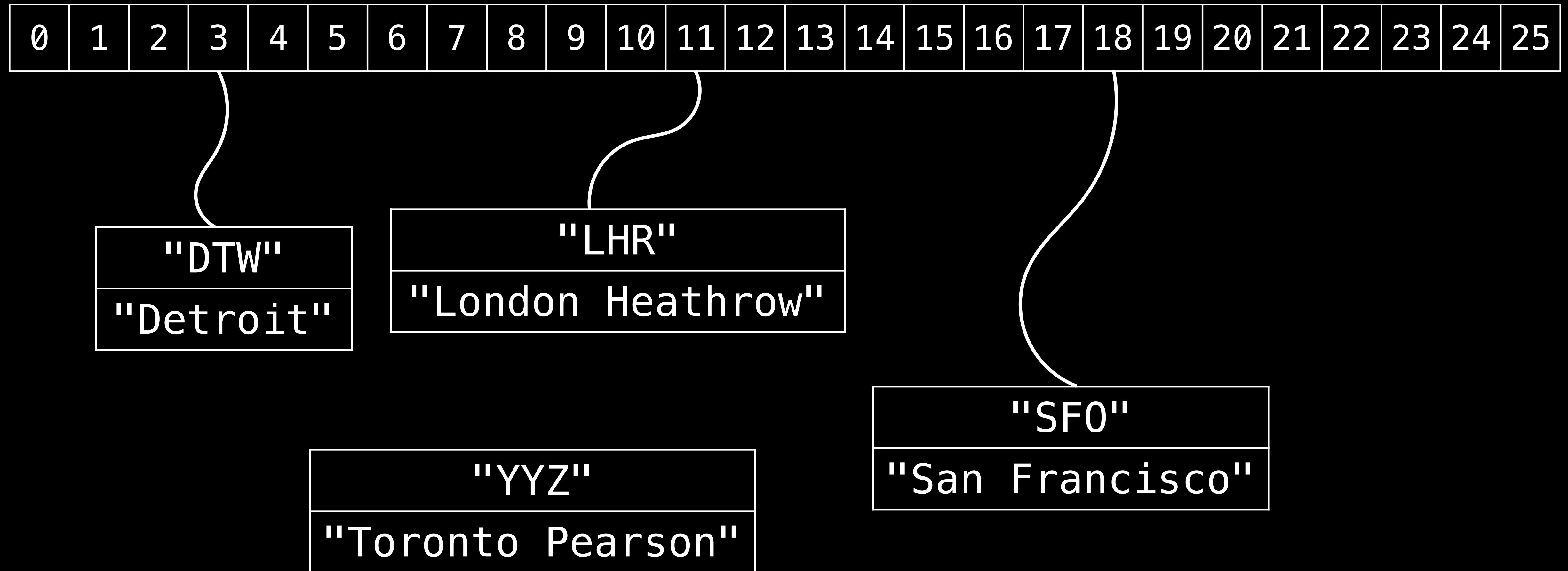
Linear probing



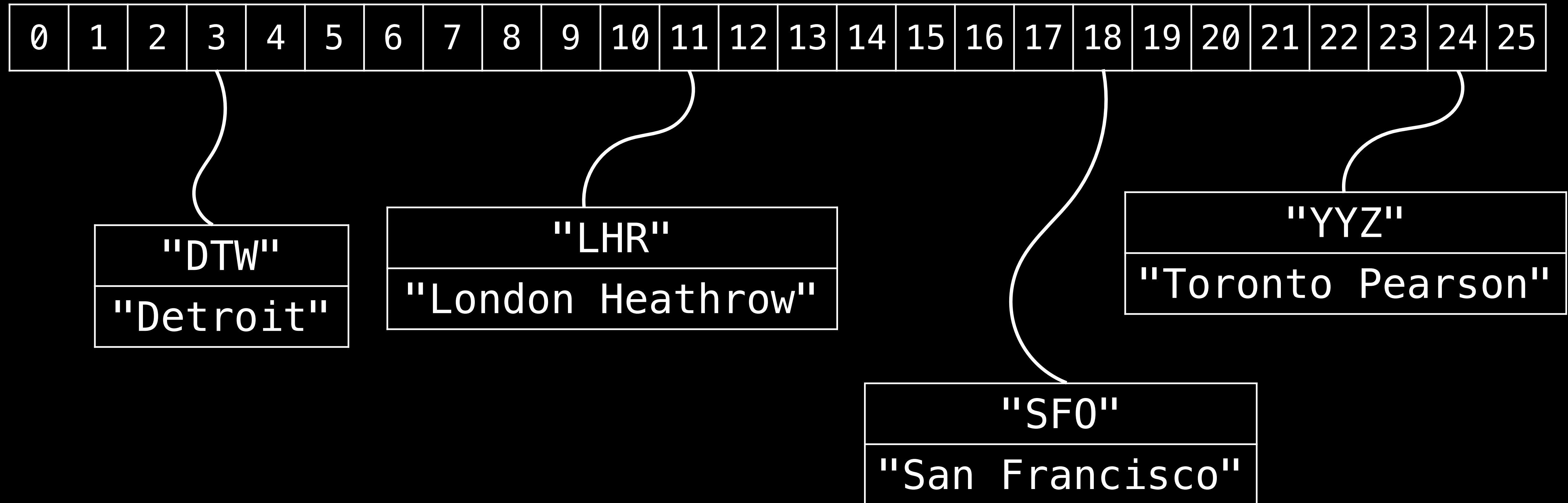
Linear probing



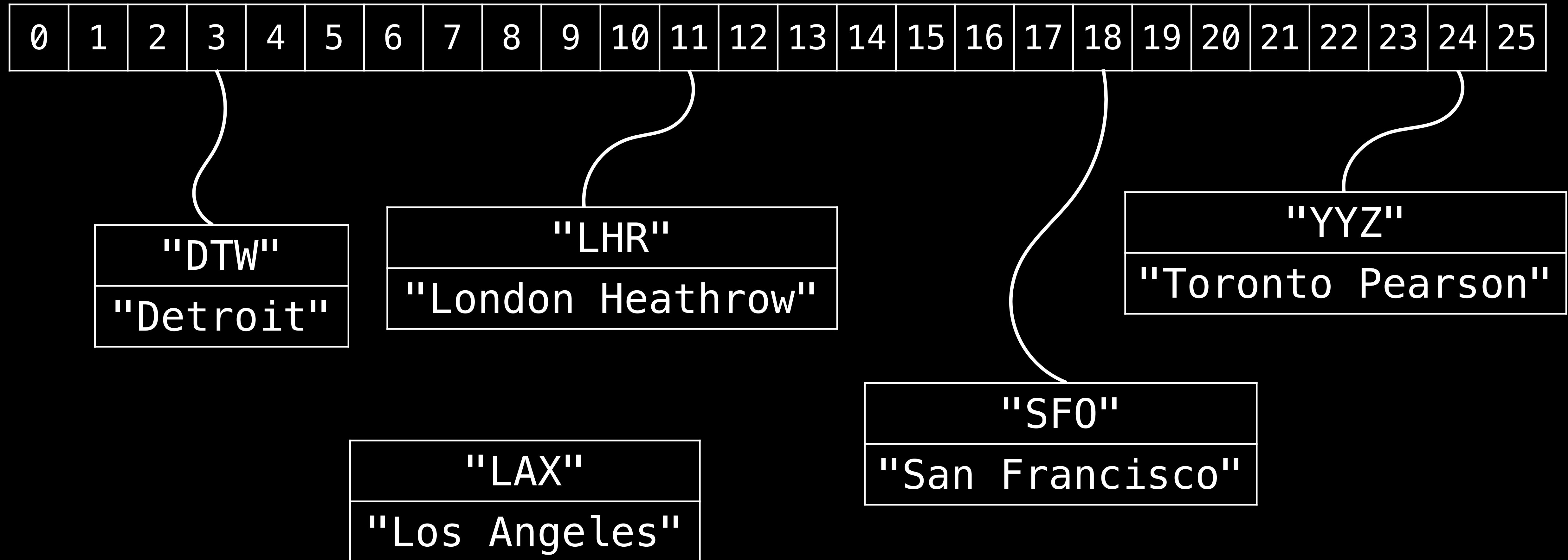
Linear probing



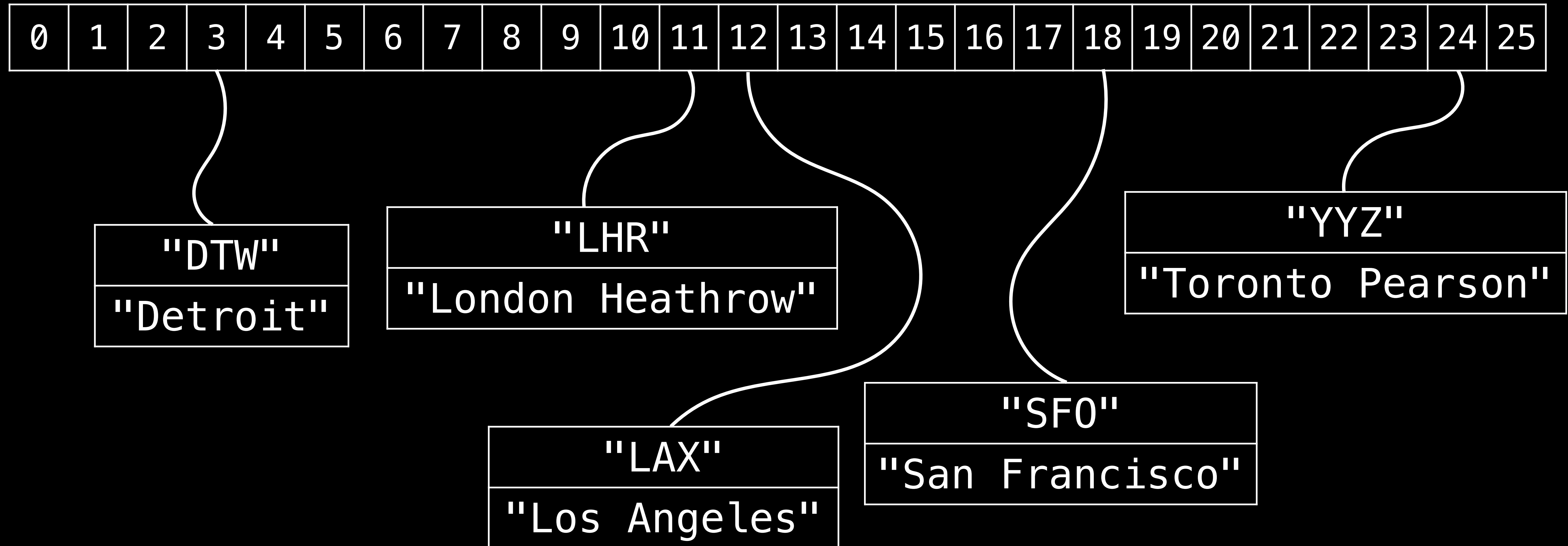
Linear probing



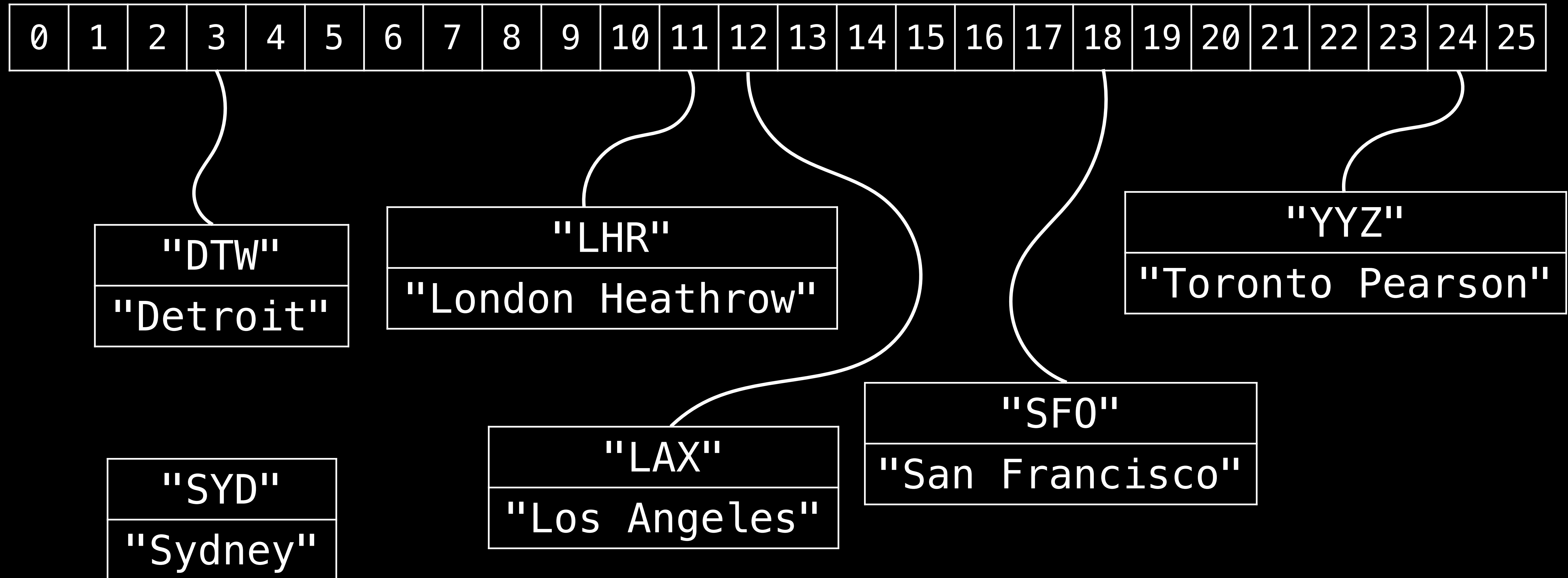
Linear probing



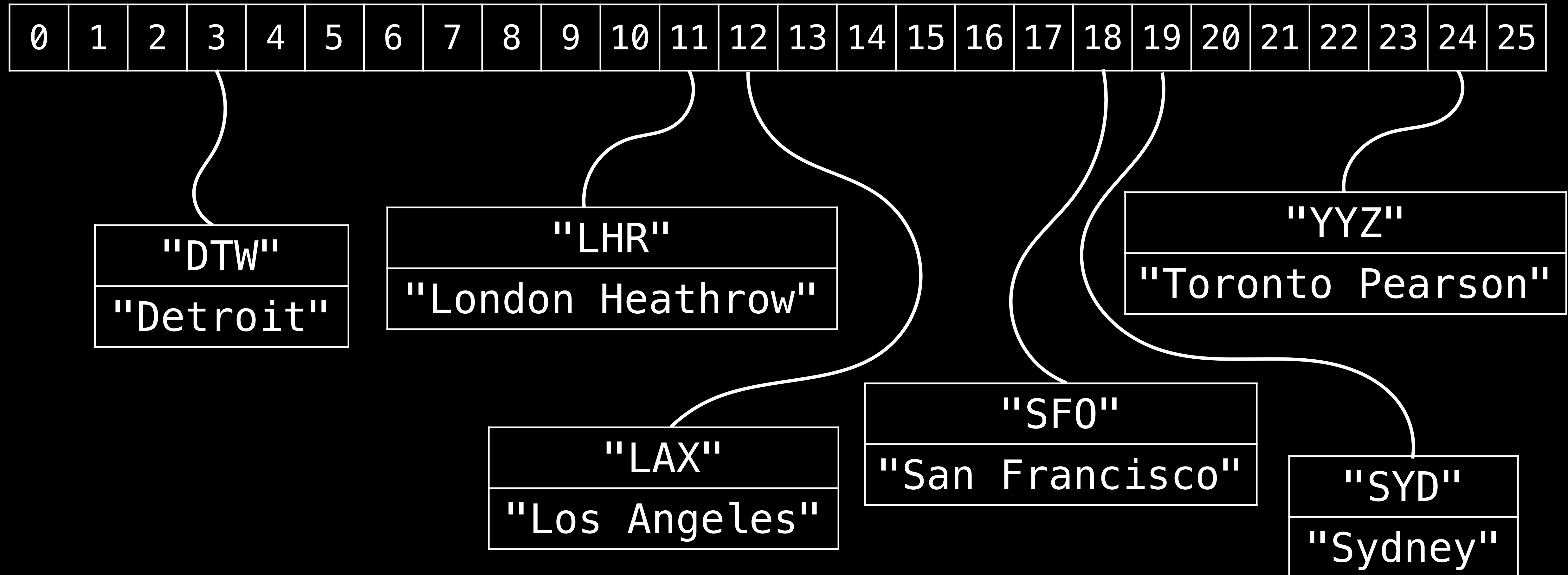
Linear probing



Linear probing



Linear probing



Chaining vs. Probing

Chaining

Handling collisions is faster

Memory indirection faster
than math + reindexing

Uses more dynamic memory

Probing

No extra memory

Simpler storage

Load factor

Load factor

$$\alpha = \frac{n}{m}$$

← # of elts stored

← size of array

ideally $\alpha < 1$

Chaining

exp. time $\Theta(1 + \alpha)$

Hash tables with no collisions

① Hash table where $|array| = |U|$
✓ good for char

↳ problem with ints
 $-2^{31} \dots + 2^{31}$

But values are only 0-100

int scoreCounts[101]

store # ppl w that score

Dynamic Hashing

Dynamic Hashing

resizing array when α is not small

- Similar to resing vector
- Double the size
- insertion is $\Theta(1)$ amortized expected

Ex: $n = 10$ $m = 16$ $\rightarrow$ $n = 20$ $m = 32$ $\rightarrow$ $n = 5$ $m = 8$

Ex: Double m when $n \geq m$
Shrink when $n < \frac{1}{2} m$

n	10	15	16	17	18	16
m	16	16	16	32	32	16

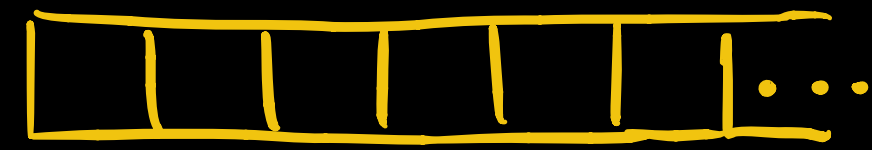
$15 - 16 - 17 - 16 - 17$
 $16 \leftrightarrow 32$ is bad

STL Containers

STL Containers

① Sequence Containers

Vector



Deque



List



Forward List



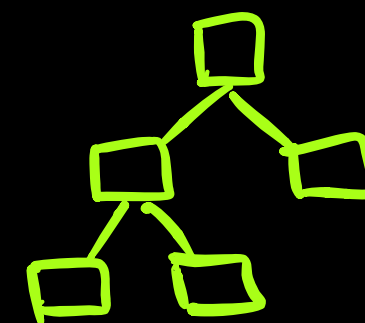
Array



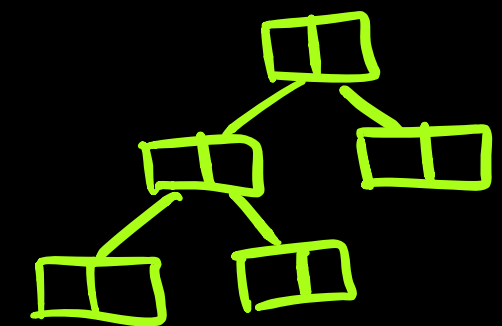
(String)

② Associative Containers

(Multi)set

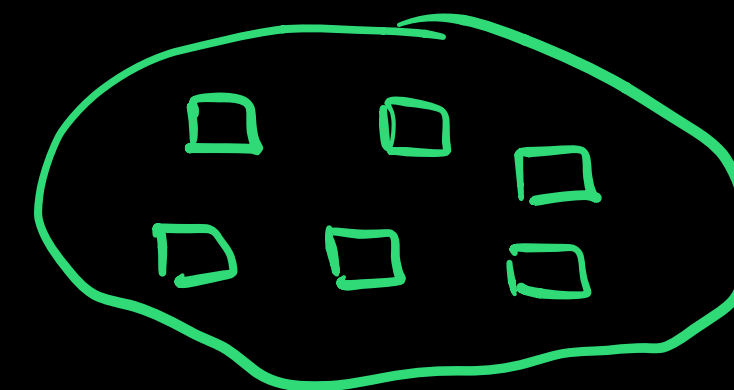


(Multi)map

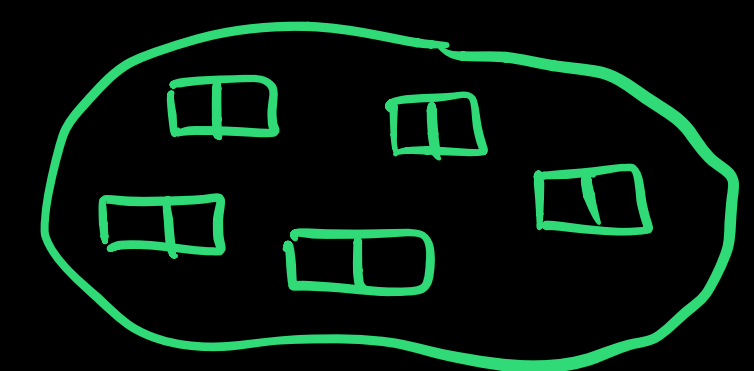


③ Unordered Associative Containers

Unordered (multi)set



Unordered (multi)map



④ Container Adapters

stack, queue, priority queue

STL Containers

① Sequence Containers

Vector



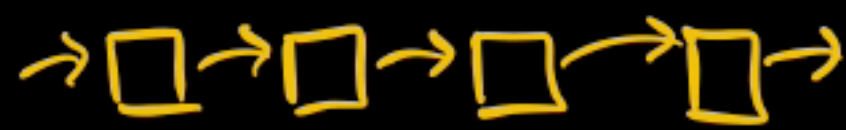
Deque



List



Forward List



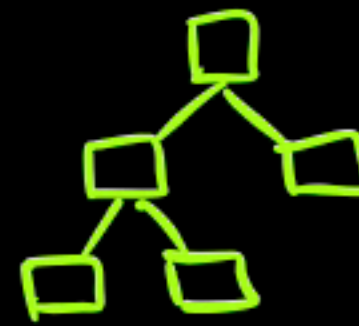
Array



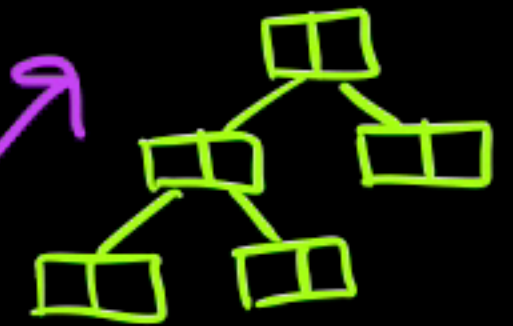
(String)

② Associative Containers

(Multi)set



(Multi)map

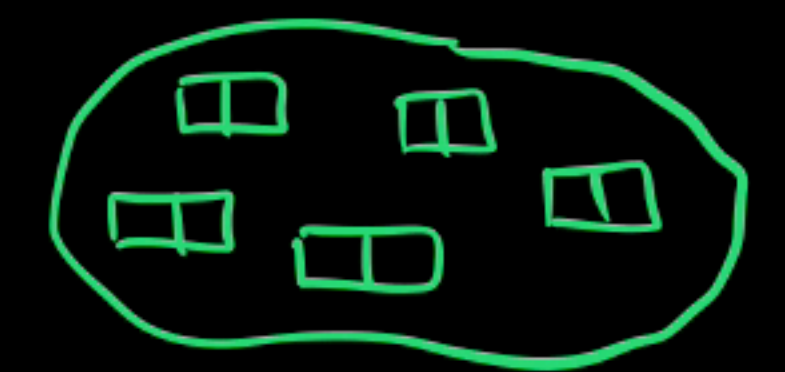
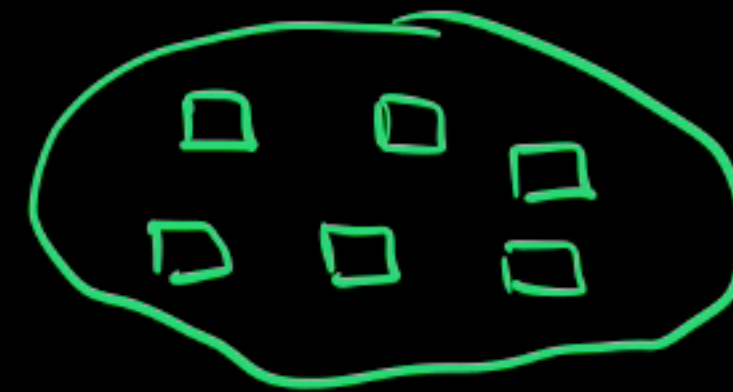


key-value pairs

③ Unordered Associative Containers

Unordered (multi)set

Unordered (multi)map



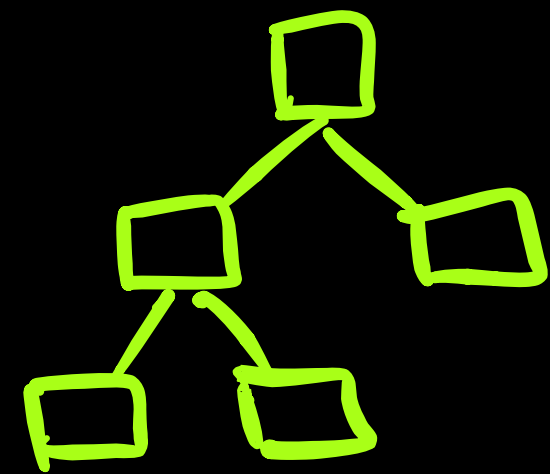
④ Container Adapters

stack, queue, priority queue

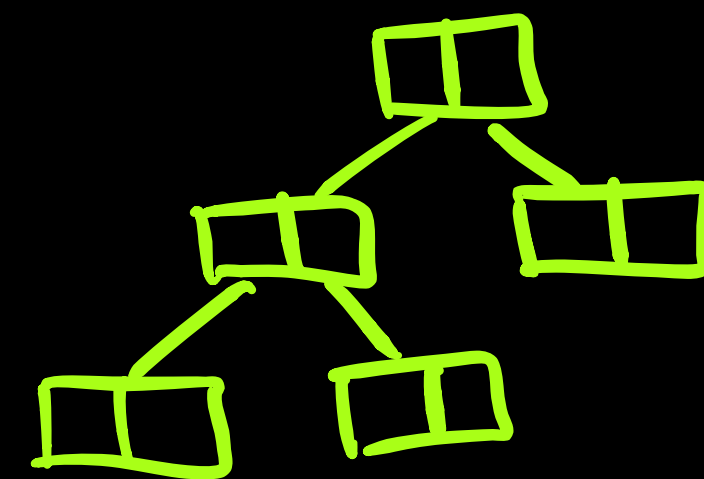
sorted
($\angle$ by default)

Associative containers

(Multi)set

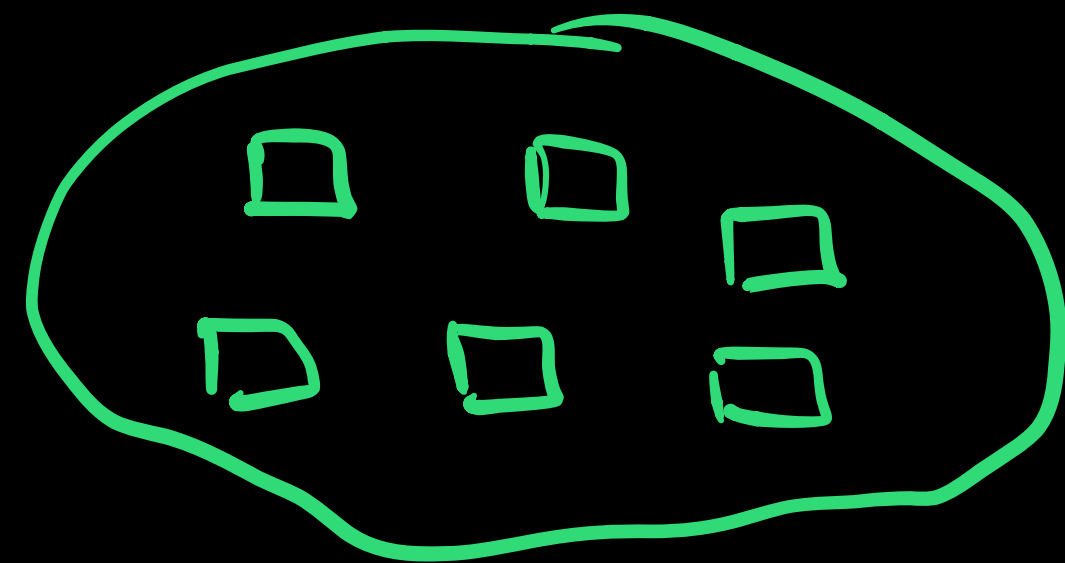


(Multi)map)

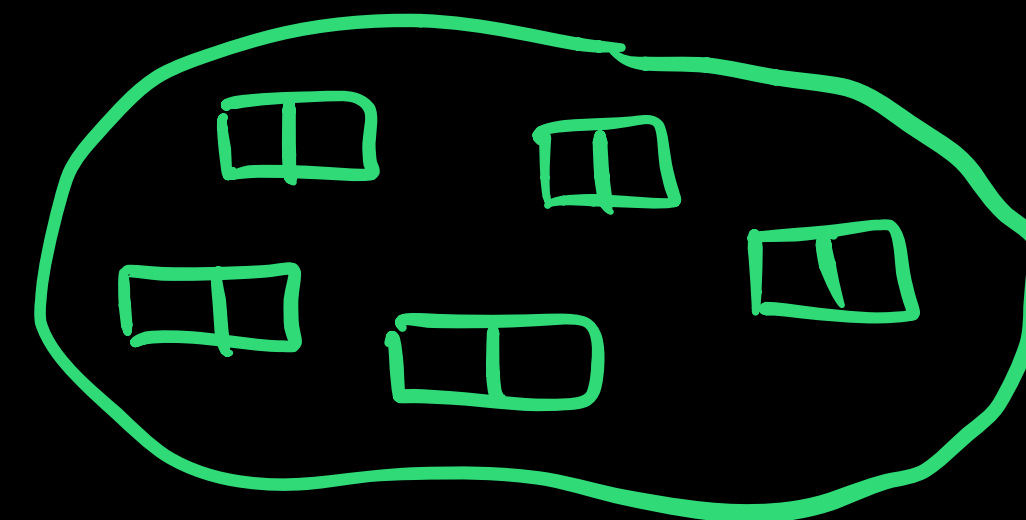


Unordered containers

Unordered (multi)set

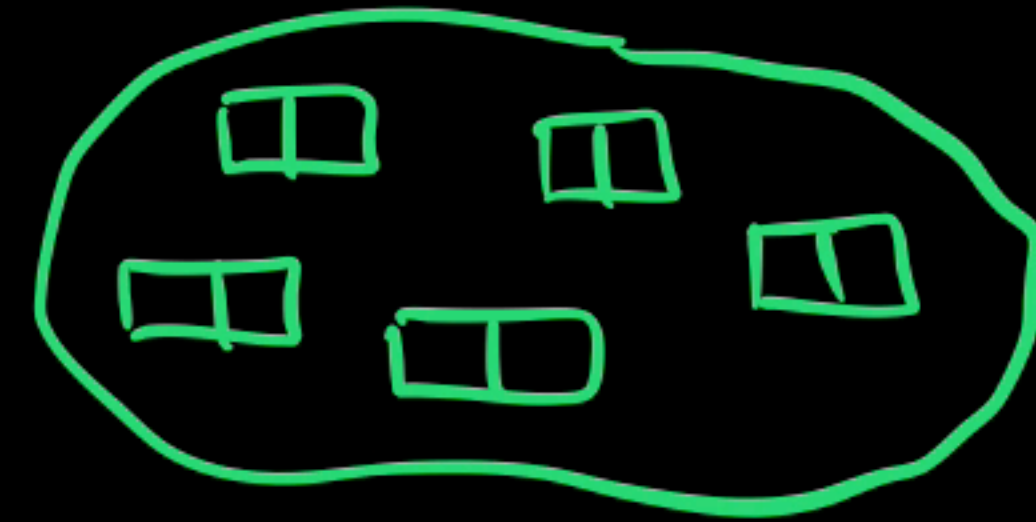
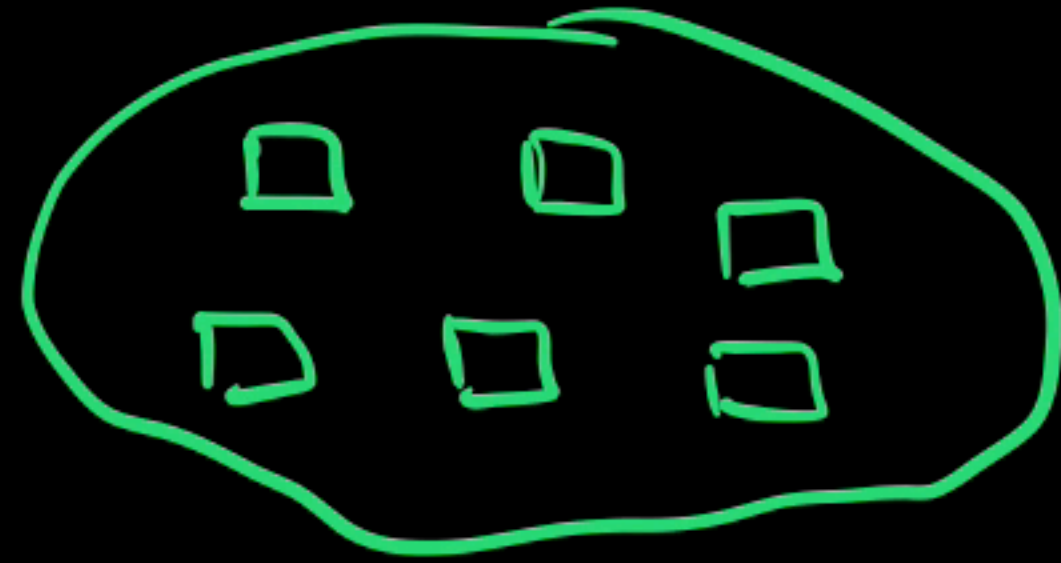


Unordered (multi)map



Unordered containers

Unordered (multi)set Unordered (multi)map



Hash tables

Exp.

$\Theta(1)$

insert

find

remove (erase)

Unordered set

```
unordered_set<string> colors = {"red", "blue", "white"};

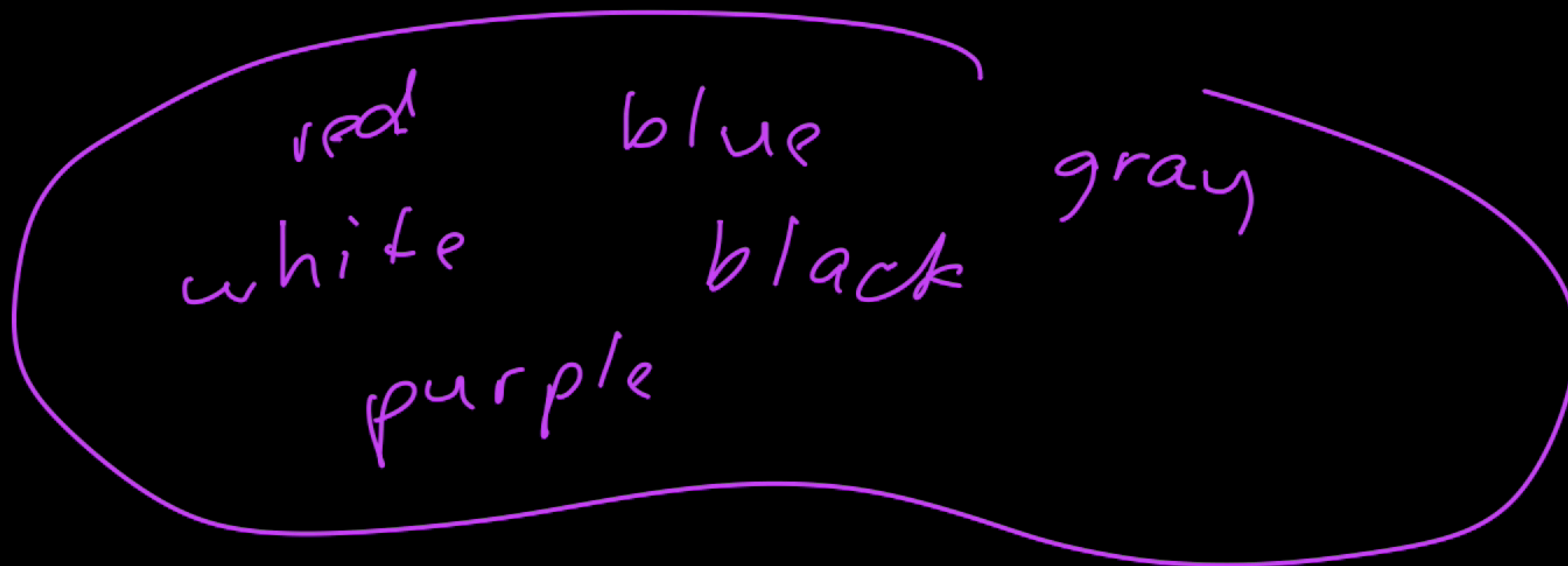
auto it2 = colors.find("blue");
if (it2 == colors.end()) {
    // important!
    cout << "not found" << endl;
} else {
    cout << *it << endl;
}
```

Unordered set

```
colors.insert("black");  
vector<string> moreColors = {"purple", "gray"};  
colors.insert(moreColors.begin(), moreColors.end());
```

Unordered set

```
colors.insert("black");  
vector<string> moreColors = {"purple", "gray"};  
colors.insert(moreColors.begin(), moreColors.end());
```



Unordered set

```
for (auto it = colors.begin(); it != colors.end(); ++it)
{
    cout << *it << endl;
}
```

```
for (auto &color: colors) {
    cout << color << endl;
}
```

Unordered set

cannot do <

```
for (auto it = colors.begin(); it != colors.end(); ++it)
{
    cout << *it << endl;
}
```

order is not
deterministic

```
for (auto &color: colors) {
    cout << color << endl;
}
```

Unordered map

```
unordered_map<string, string> colors = {
    {"white", "#ffffff"},
    {"blue", "#0000ff"},
    {"red", "#ff0000"}
};

if (it == colors.end()) {
    // important!
    cout << "not found" << endl;
} else {
    cout << it->first << " is " << it->second << endl;
}
```


Unordered map

RGB hex
each 0...255
used in CSS

```
unordered_map<string, string> colors = {  
    {"white", "#ffffff"},  
    {"blue", "#0000ff"},  
    {"red", "#ff0000"}  
};  
auto it = colors.find("red");  
if (it == colors.end()) {  
    // important!  
    cout << "not found" << endl;  
} else {  
    cout << it->first << " is " << it->second << endl;  
}
```

init list
for pairs

red #ff0000

Unordered map

```
colors.insert(pair<string, string>("blue", "#0000ff"));
```

```
colors.insert(make_pair("blue", "#0000ff"));
```

```
colors.insert({"blue", "#0000ff"});
```

Unordered map

```
colors.insert(pair<string, string>("blue", "#0000ff"));
```

```
colors.insert(make_pair("blue", "#0000ff"));
```

```
colors.insert({"blue", "#0000ff"});
```

auto it will fail after first insertion
return iterator to what is in map

Associative array

```
colors["blue"] = "#0000ff";  
// equivalent to  
// colors.insert({"blue", "#0000ff"});
```

```
colors["blue"] = "#00274c";  
// changes the value unlike insert()
```

```
cout << colors["blue"] << endl;
```

Associative array

[]
↖ not nec. int's

```
colors["blue"] = "#0000ff";  
// equivalent to  
// colors.insert({"blue", "#0000ff"});
```

```
colors["blue"] = "#00274c";  
// changes the value unlike insert()
```

succeeds after
1st insertion

```
cout << colors["blue"] << endl;
```

CAREFUL when not in map!

~~* (colors.end())~~

""

Associative array

```
void darken(const unordered_map<string, string> &c) {  
    // will not compile  
    c["white"] = "#eeeeeee";  
}  
  
// ...  
  
darken(colors);
```


Associative array

```
void darken(const unordered_map<string, string> &c) {  
    // will not compile  
    c["white"] = "#eeeeee";  
}
```

```
// ...
```

```
darken(colors);
```

X error

Associative array

```
void print(const unordered_map<string, string> &c) {  
    // will not compile  
    cout << c["white"] << endl;  
}  
  
// ...  
  
print(colors);
```

Associative array

```
void print(const unordered_map<string, string> &c) {  
    // will not compile  
    cout << c["white"] << endl;  
}
```

```
// ...
```

```
print(colors);
```

write
access

X error

Associative array

```
void print(const unordered_map<string, string> &c) {  
    auto it = c.find("white");  
    if (it != c.end()) {  
        cout << it->second << endl;  
    }  
}  
  
// ...  
  
print(colors);
```

Associative array

```
void print(const unordered_map<string, string> &c) {  
    auto it = c.find("white");  
    if (it != c.end()) {  
        cout << it->second << endl;  
    }  
}  
  
// ...  
print(colors);
```

it to pair

*pair . first
 . second*



Cryptographic hash functions

Cryptographic hash functions

Digital signatures

Software updates

Checksums

Backup systems

Git

Storing passwords

...

Cryptographic hash functions

Deterministic

Quick to compute for any given message

Infeasible to get a key from its hash value except by brute force

A small change to a key should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value

Infeasible to find two different keys with the same hash value

Cryptographic hash functions

✓ Deterministic

✓ Quick to compute for any given message

arbitrary string

160 bits
256 bits

poly-time
okay

MD1

MD5

SHA1

SHA3

SHA256

Infeasible to get a key from its hash value except by brute force

A small change to a key should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value

Infeasible to find two different keys with the same hash value

Storing passwords

Storing passwords

salt	db	user	password
		trevor	12345 $\text{hash}_1(12345)$
		trevor2	$\text{hash}_2(123215)$

Q&A