# ABSTRACT DATATYPES

ADS1, S2023

# ABSTRACT DATATYPES
(ADTs)

are objects with a set of operations, but we don't care about how it's actually implemented.

ADTs

List
Stack
Queues


DS → data structure

linked List
Array List

# WE DISTINGUISH

ADT

Data objects for a problem

from

DS

Their representation in memory

# ABSTRACT DATATYPES IN JAVA

ADT

```
public interface ADSListADT<T> {

    public void add(T elm);

    public void insert(int index, T elm);

    public boolean remove(T elm);

    public int indexOf(T elm);

    public int size();

    public boolean contains(T elm);

    public T remove(int index);

    public T set(int index, T elm);

}
```

```
public class ADSLinkedList<T> implements
ADSListADT<T> {

    private Node<T> first;
    private int size;

    public ADSLinkedList(){
        first = null;
        size = 0;
    }

    @Override
    public void add(T elm) {
        if(elm == null){
            return;
        }
        Node<T> newNode = new Node(elm, null);
        if(size == 0){
            first = newNode;
        ... blah blah blah …
```

# SETS, MAPS AND HASHING

ADS1, S2023

# SETS

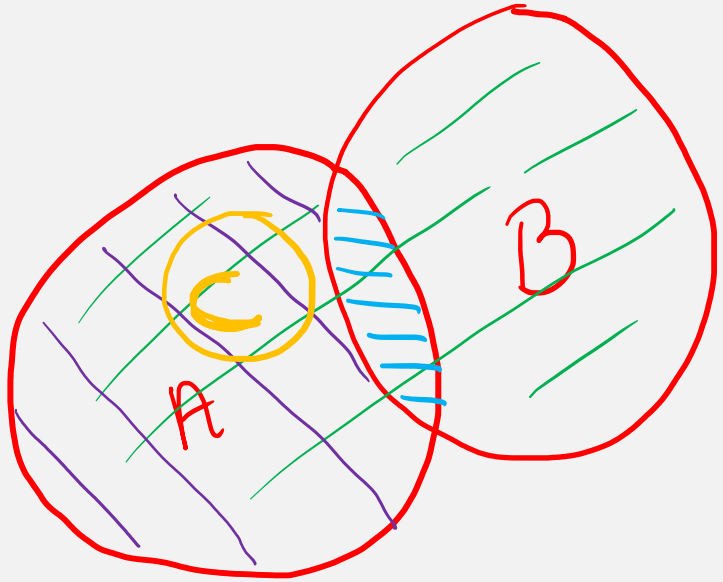
A collection of elements with no duplicates and no order

$S = \{\text{"apple"}, \text{"pear"}, \text{"banana"}\}$

add "apple" to $S$ → nothing happens

# OPERATIONS ON A SET INCLUDE …

- … testing for membership
- … adding elements
- … removing elements
- … union $A \cup B$
- … intersection $A \cap B$
- … set difference $A - B$
- … subset $C \subseteq A$



# A SMALL EXERCISE

$A = \{1, 3, 5, 7\}$ $B = \{2, 3, 4, 5\}$

Find

$A \cup B = \{1, 2, 3, 4, 5, 7\}$

$A \cap B = \{3, 5\}$

$A - B = \{1, 7\}$

$B - A = \{2, 4\}$

# LISTS VS SETS

Sets …

have no duplicates

have no positions

⇒ can give average constant time complexity for lookup, insert, delete

# MAPS aka DICTIONARIES

sets of ordered pairs

(key, value)

efficient way of storing information

no duplicate keys

may appear several times

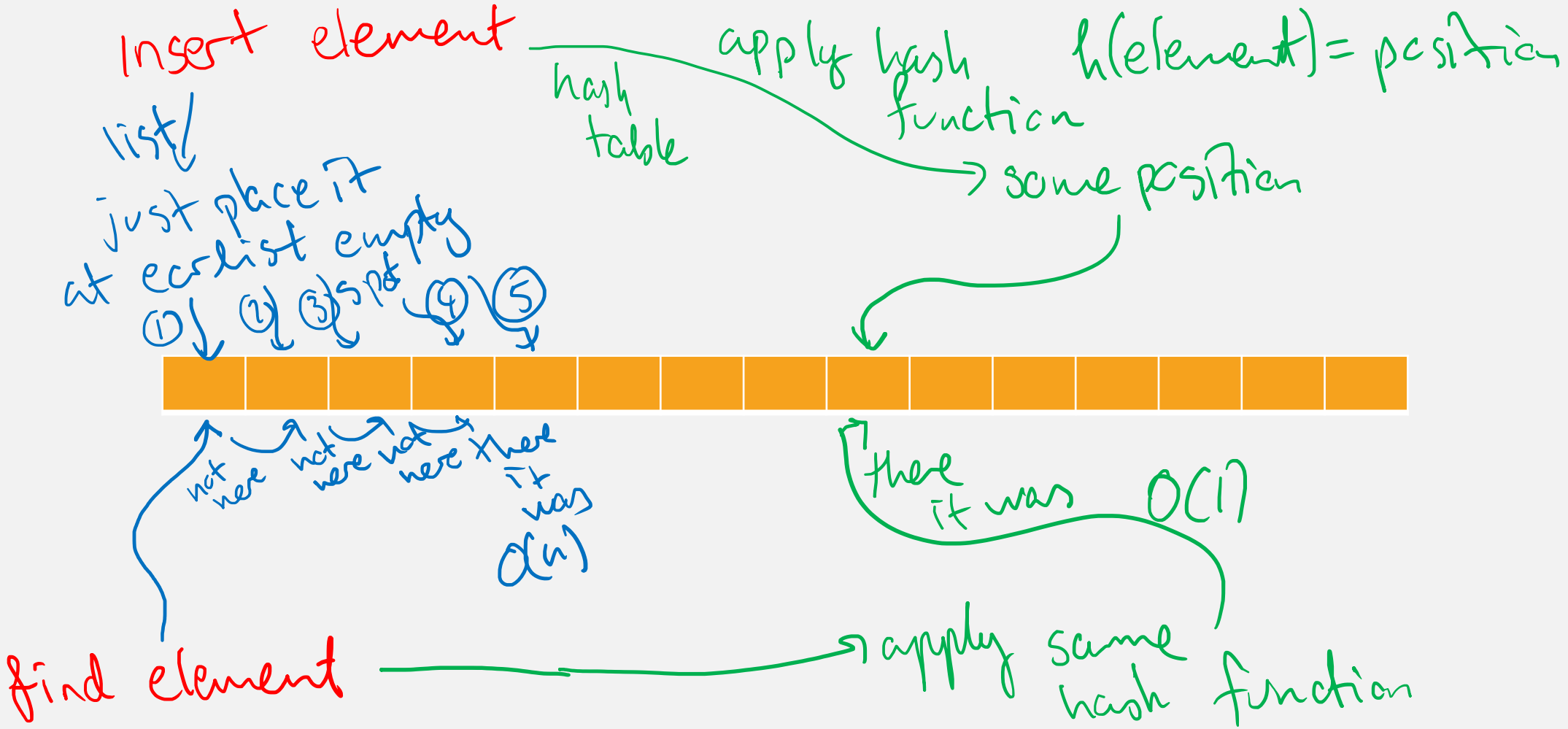
planet_radii = { (‘mercury’, 2440), (‘venus’, 6052), (‘earth’, 6378), …}

# OPERATIONS ON A MAP INCLUDE …

- … getting a value given a key (*get*)
- … adding a pair (key, value) (*put*)

how to implement sets & maps?

# HASH TABLES

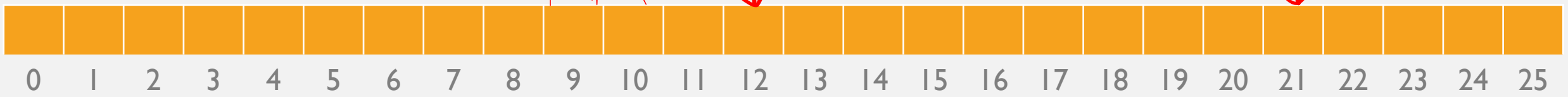

insert element
apply hash function
h(element) = position
hash table
some position
list
just place it at end of list
1
2
3
spot
4
5
not here
not here
not here
there it was
O(1)
there it was
O(1)
find element
apply same hash function

# A SET OF PLANETS

hash fct = position in alphabet of first letter in planet name

h("Mercury") = 12
h("Venus") = 21
h("Earth") = 4
h("Mars") = 12


COLLISION



# THE JAVA STRING HASH FUNCTION

unicode char of last letter $\times 31^0$

\+ ——— 2nd last ——— $\times 31^1$

\+ ——— 3rd last ——— $\times 31^2$

\+ . . . . . .

$h(\text{"Earth"}) = \text{"h"} \times 31^0 + \text{"t"} \times 31^1 + \text{"r"} \times 31^2 + \text{"a"} \times 31^3 + \text{"E"} \times 31^4$

$= 104 \times 1 + 116 \times 31 + 114 \times 31^2 + 97 \times 31^3 + 69 \times 31^4$

$= 66725930$

# "GOOD" HASH FUNCTIONS

→ spreads values evenly

→ are cheap to compute

BUT WE STILL NEED TO HANDLE COLLISIONS

# COLLISIONS

OPEN ADDRESSING

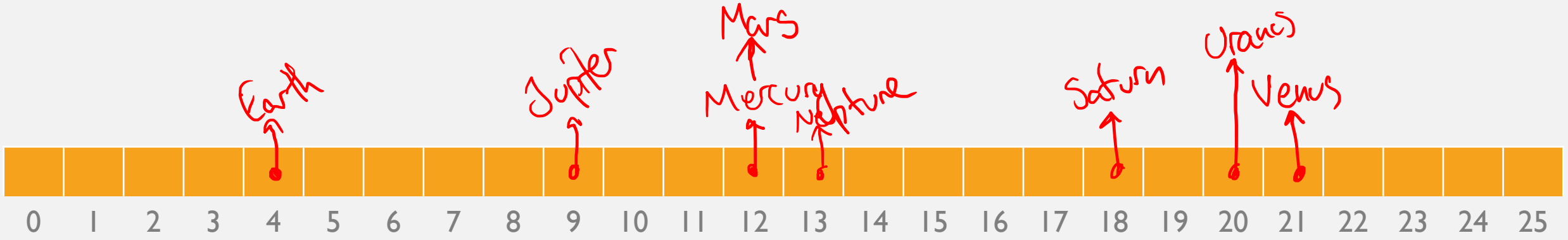
If position taken, find another one

SEPARATE CHAINING

Accept multiple items at same position

# SEPARATE CHAINING

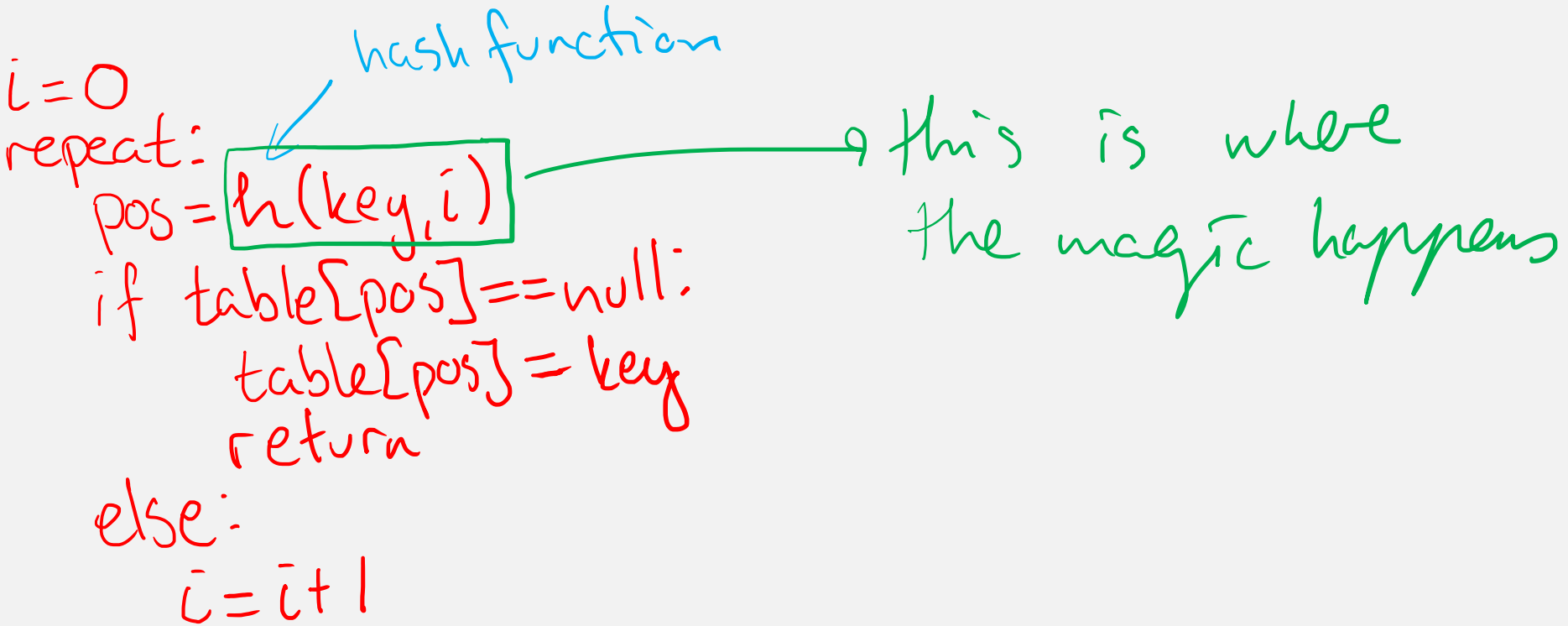
Each table slot references a linked list ("bucket")




danger: if linked lists long, no improvement at all

# OPEN ADDRESSING

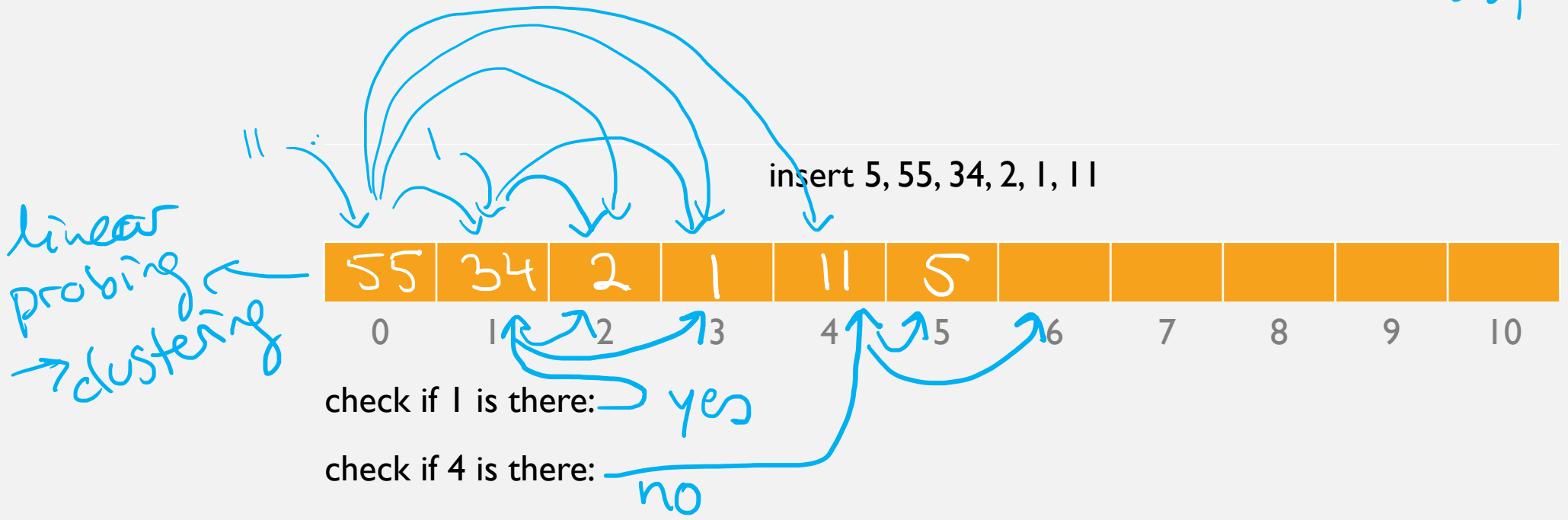
If the hashed position is occupied, find a new one



# OPEN ADDRESSING I: LINEAR PROBING

$h_1(k) = k \bmod 11$

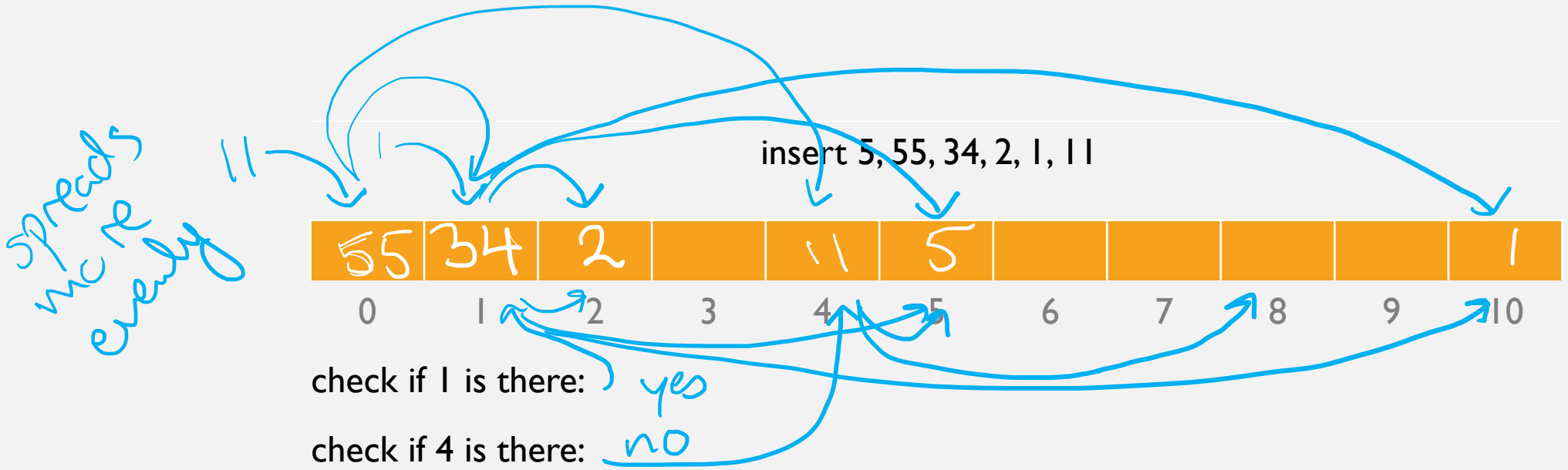
$h(k,i) = h_1(k) + i$ ← linear term { could be $2i$, $3i$, $4i$, $5i$, }



# OPEN ADDRESSING II: QUADRATIC PROBING

$h_1(x) = k \bmod 11$

$h(x) = h_1(x) + i^2$ (or some other quadratic function of $i$)

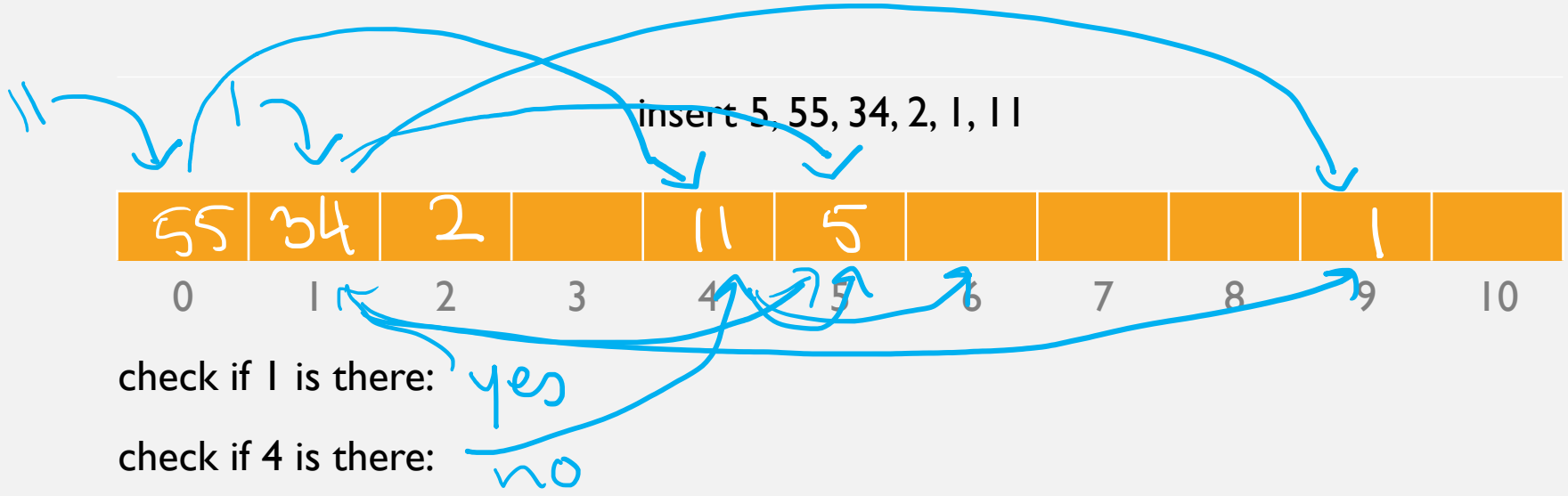


# OPEN ADDRESSING III: DOUBLE HASHING

$h_1(k) = k \bmod 11$

$h(k) = h_1(k) + \text{some other function}$

$\text{e.g.} = h_1(k) + i(5 - k \bmod 5)$



# WHY ARE HASH TABLES IDEAL FOR IMPLEMENTING SETS AND MAPS?

Hash table → no particular order

| | Hash table | Sorted array |
|---|---|---|
| Insertion | $\theta(1)$ | $O(n)$ |
| Searching | $O(1)$ | $O(\log n)$ |

# PERFORMANCE OF HASH TABLES

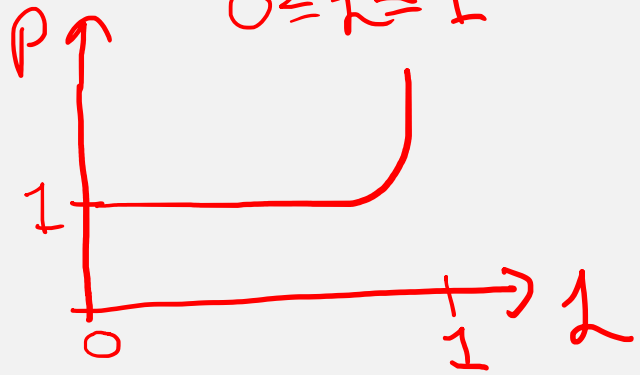
Load factor $\lambda = \frac{\#\text{filled cells}}{\#\text{cells}}$

Expected #probes $p$ for

## Linear probing

$p = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$

$0 \le \lambda \le 1$

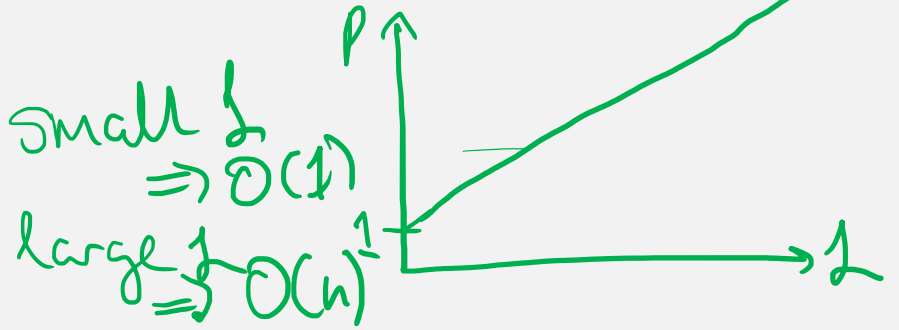



Small load factor $\Rightarrow O(1)$

## Separate chaining

$p = 1 + \frac{\lambda}{2}$

$\lambda$ can be $> 1$




Small $\lambda$ $\Rightarrow O(1)$

large $\lambda$ $\Rightarrow O(n)$