Presentation writer: Tigran Hayrapetyan

Lecturer | Programmer | Researcher

www.linkedin.com/in/tigran-hayrapetyan-cs/

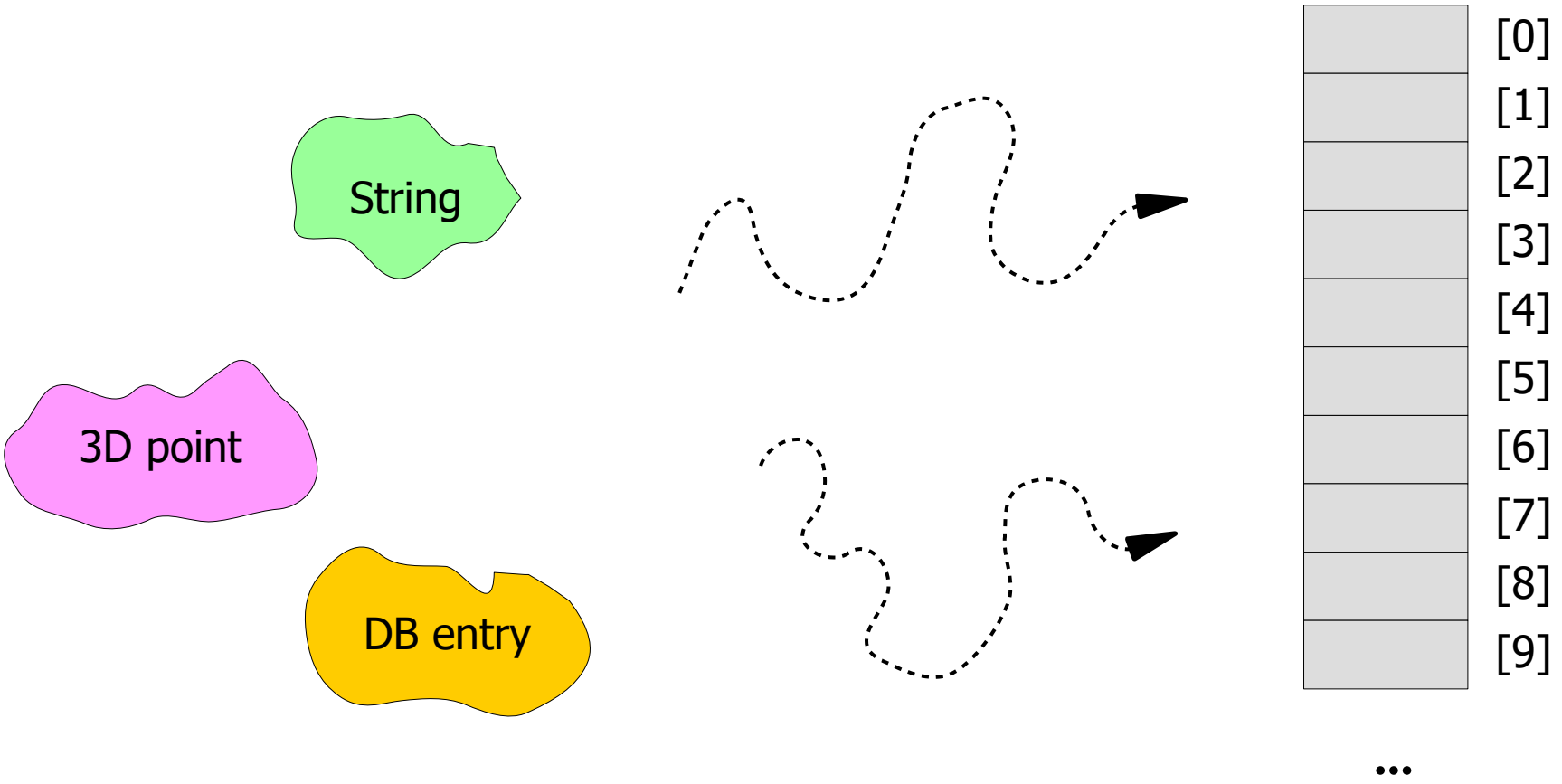
Hash table

prerequisites:

- Array,
- Linked list.

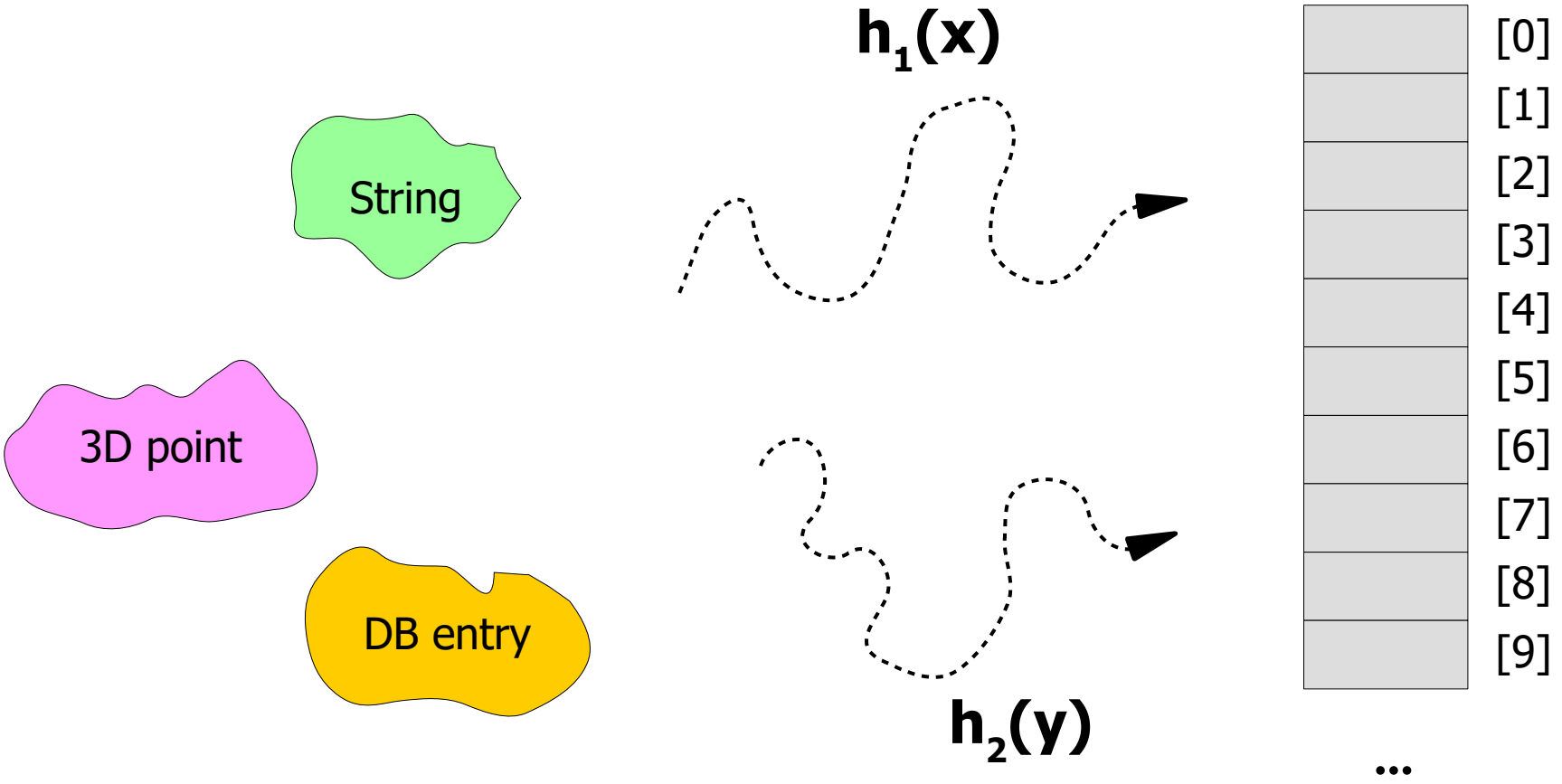
Concept of hashing

Concept of hashing is to somehow navigate complex objects into cells of an array.



Concept of hashing

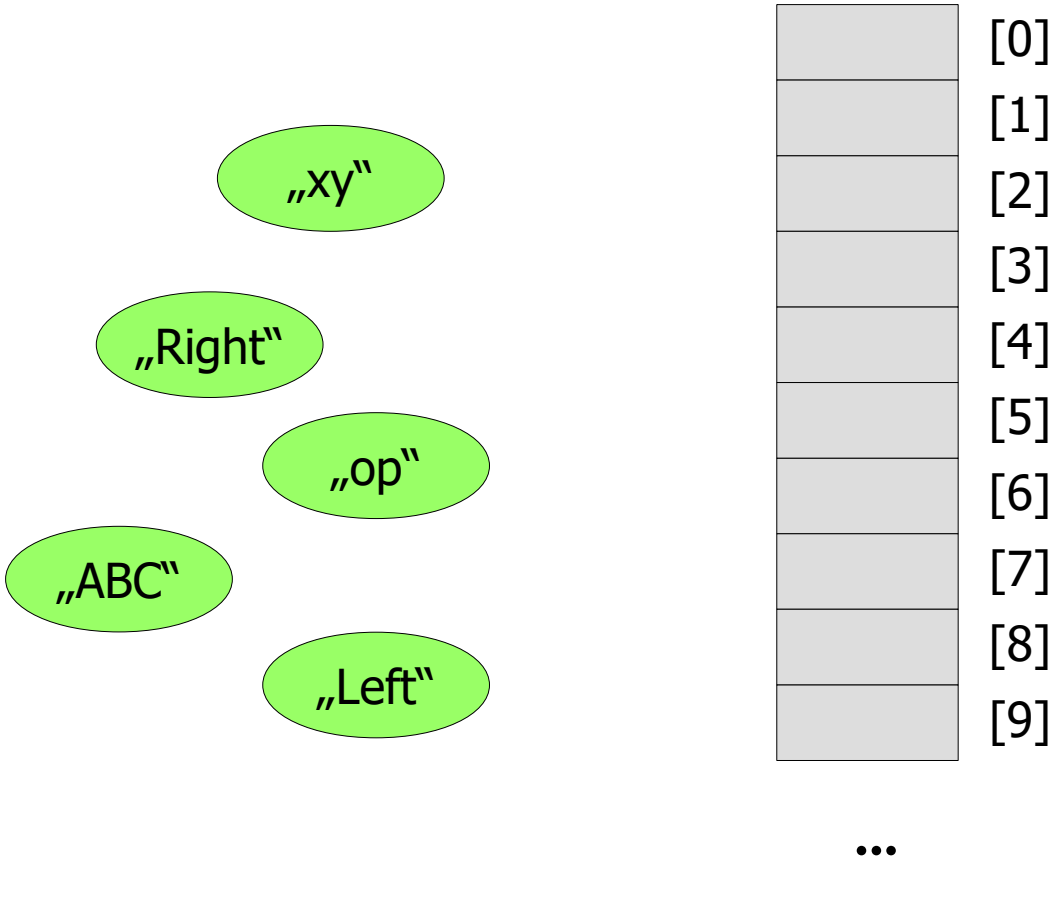
To do that, we need a hash function, defined for current type of object.



Concept of hashing

So overall concept of hash table is:

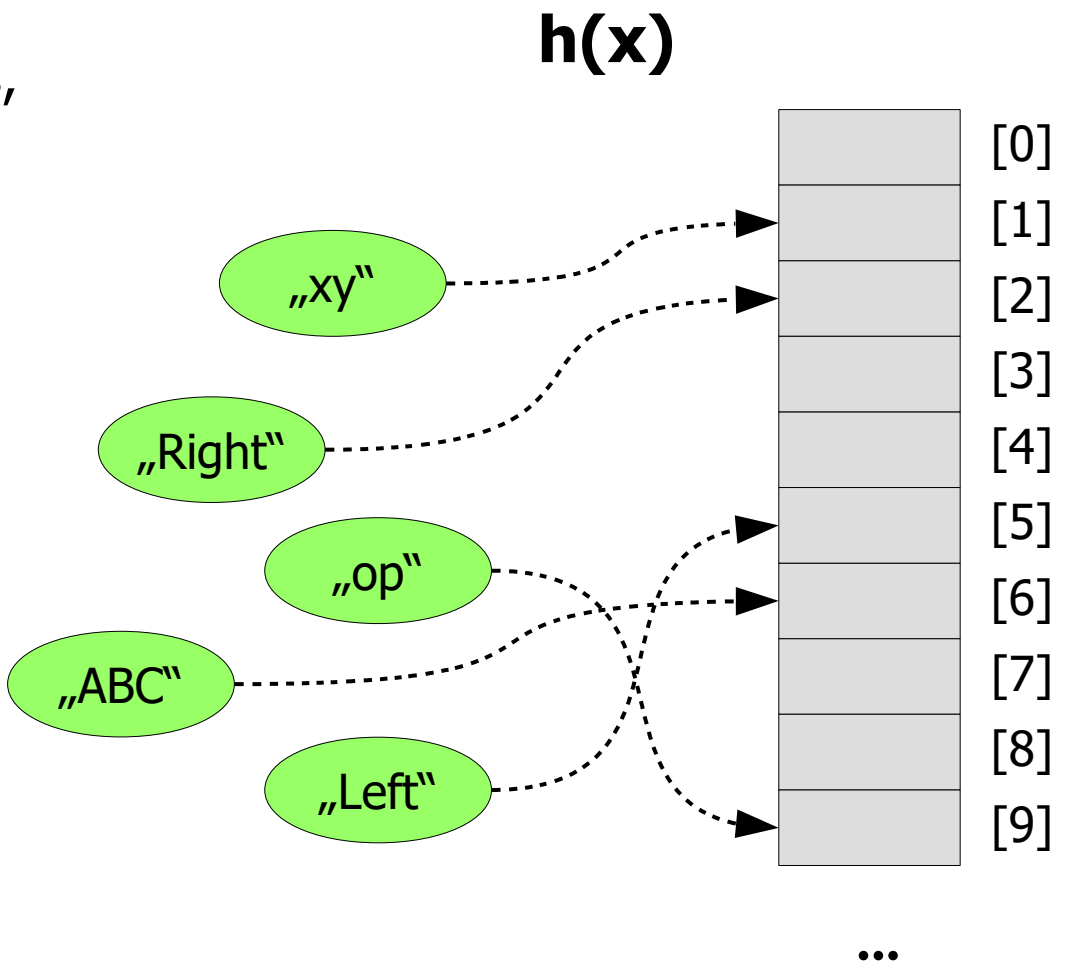
- given set of objects,



Concept of hashing

So overall concept of hash table is:

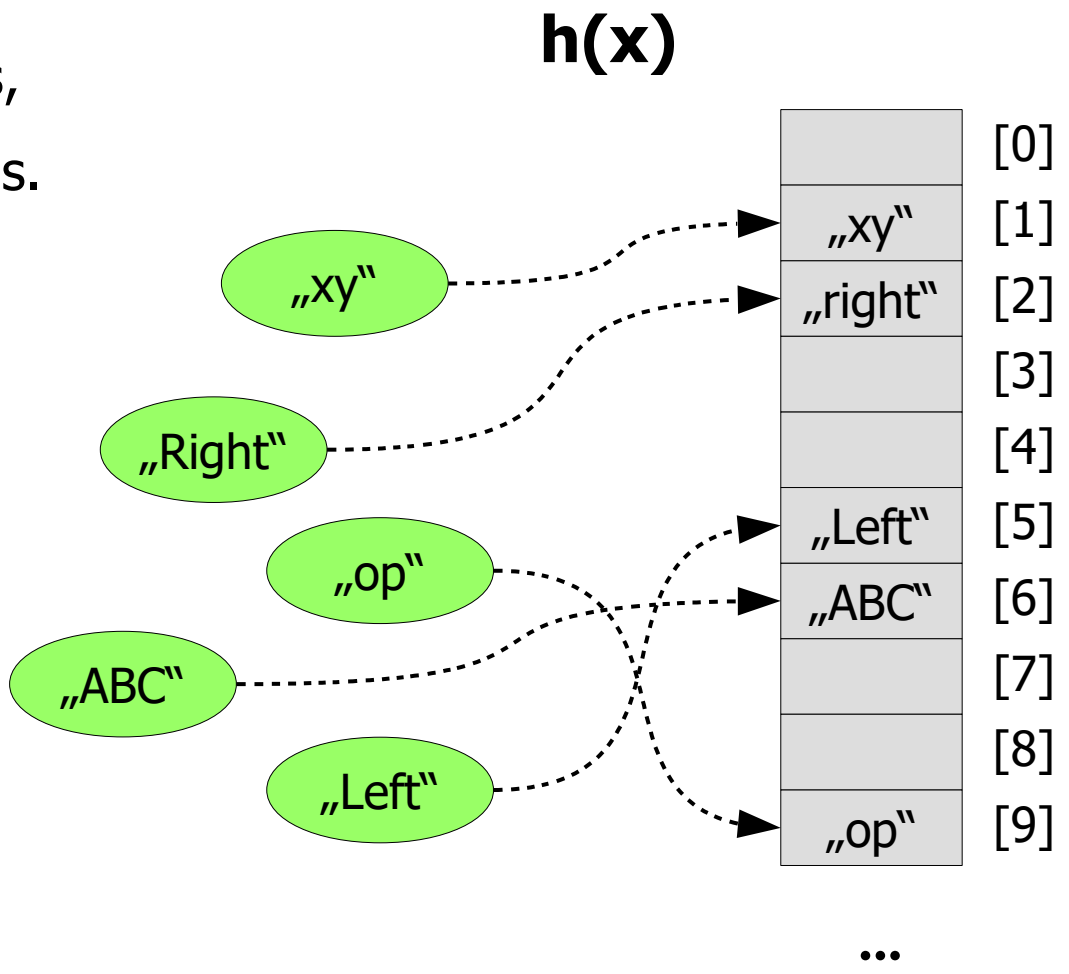
- given set of objects,
- calculate their hash values,



Concept of hashing

So overall concept of hash table is:

- given set of objects,
- calculate their hash values,
- store in corresponding cells.

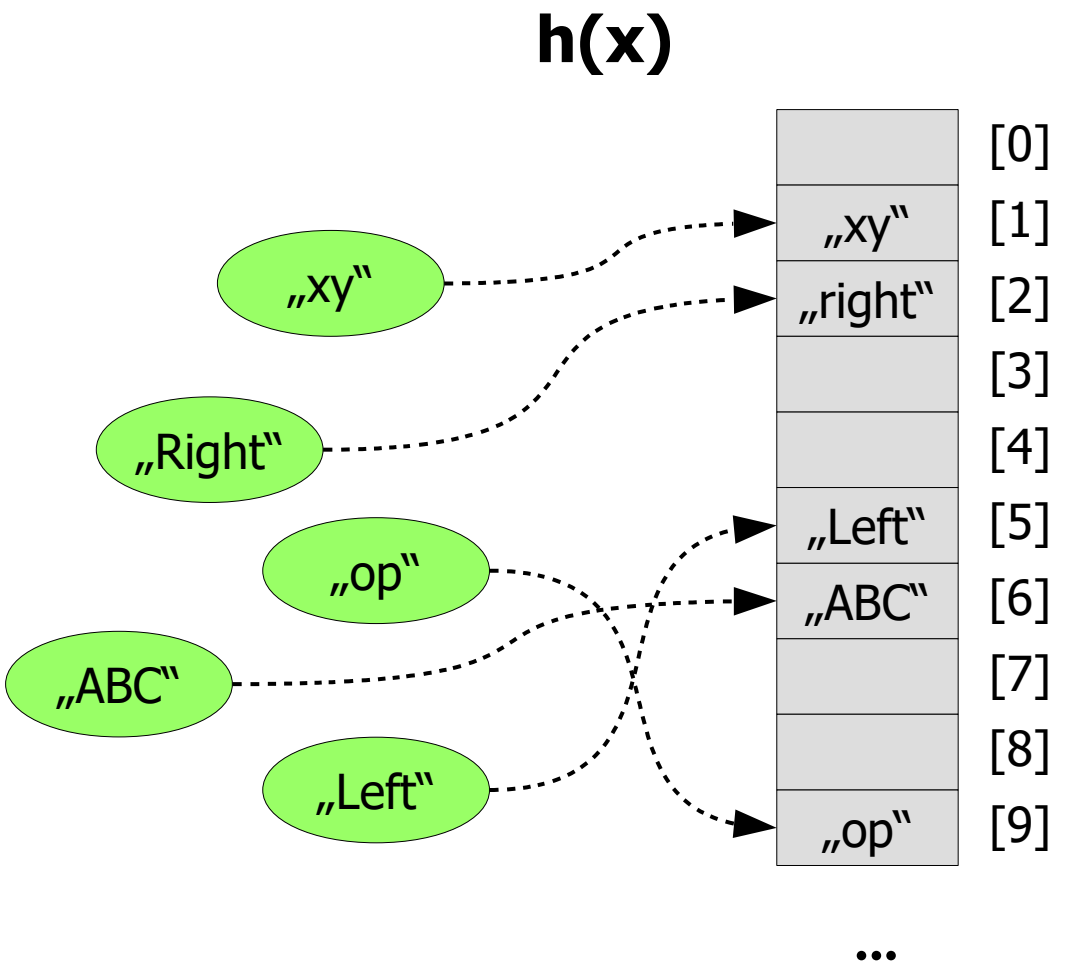


Concept of hashing

This is the most high-level concept, where:

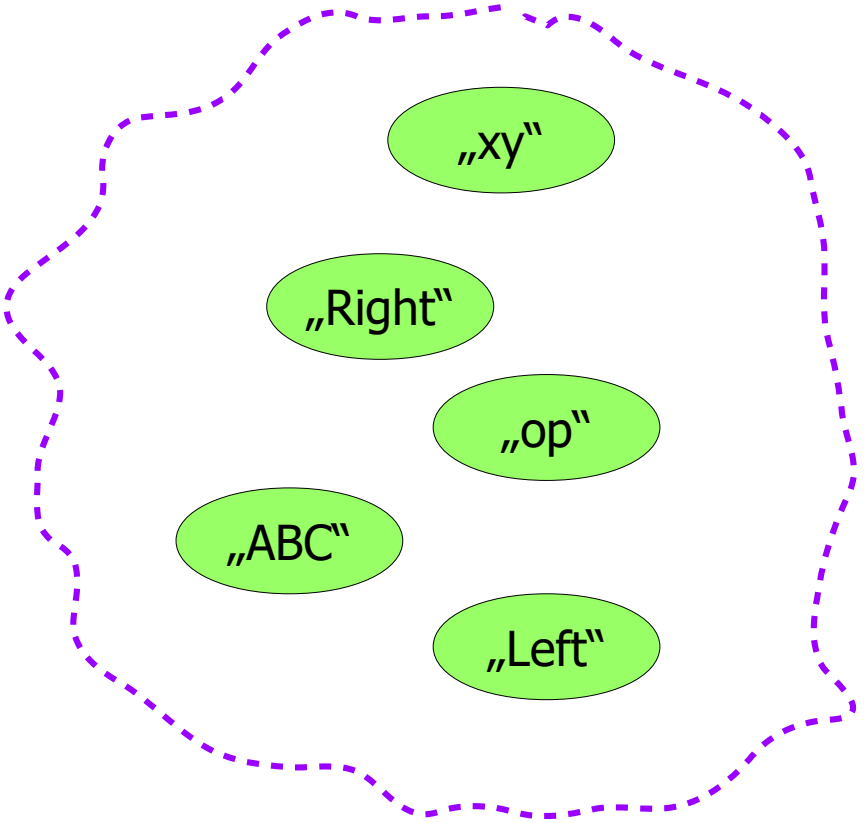
- insertions,
- searches, and
- removals

are trivial.



Concept of hashing

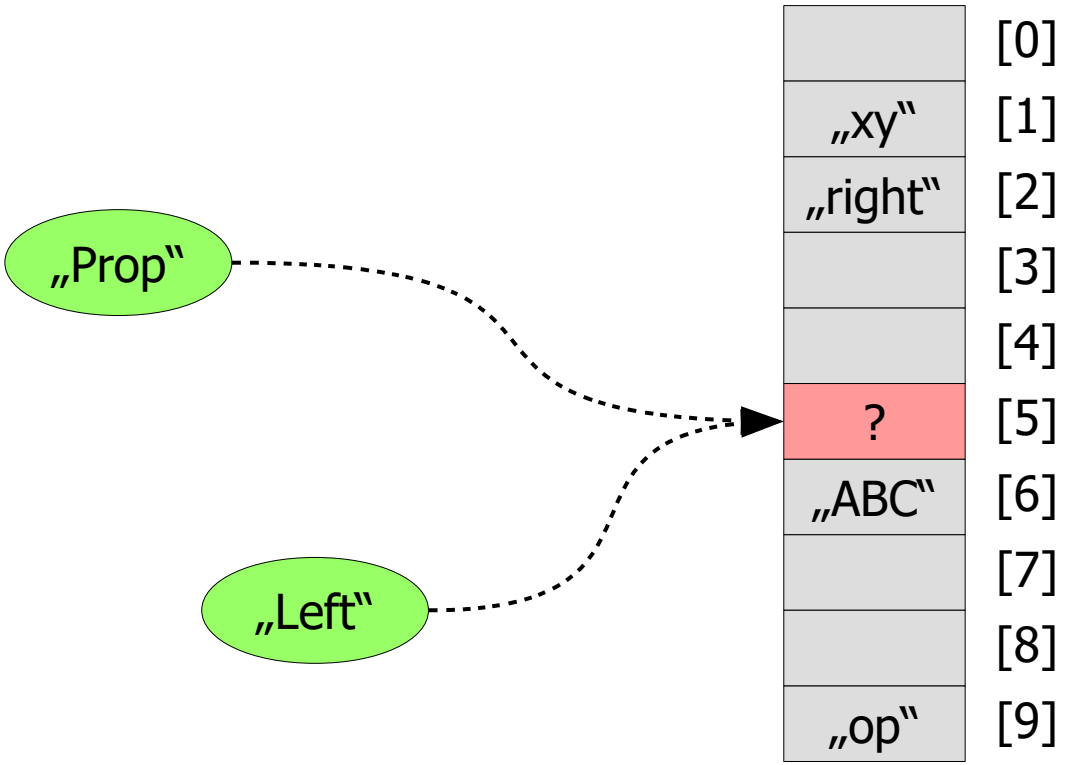
Question: What method (hash function) can you suggest for converting strings to integers?



Collisions

However, its not always that we can do that way. Because **2** different objects passed through hash function,

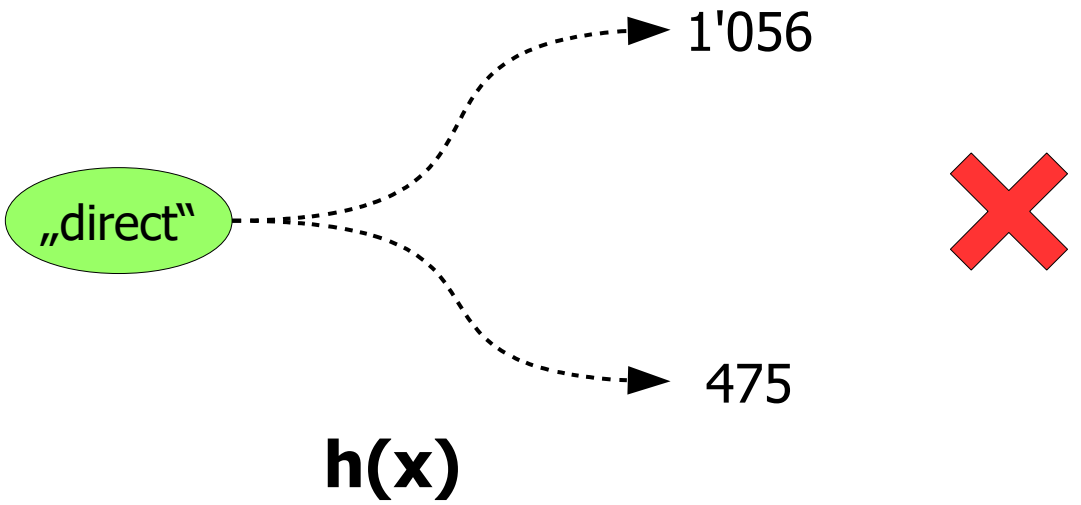
... might result in the same hash value.



This is called "collision".

Collisions

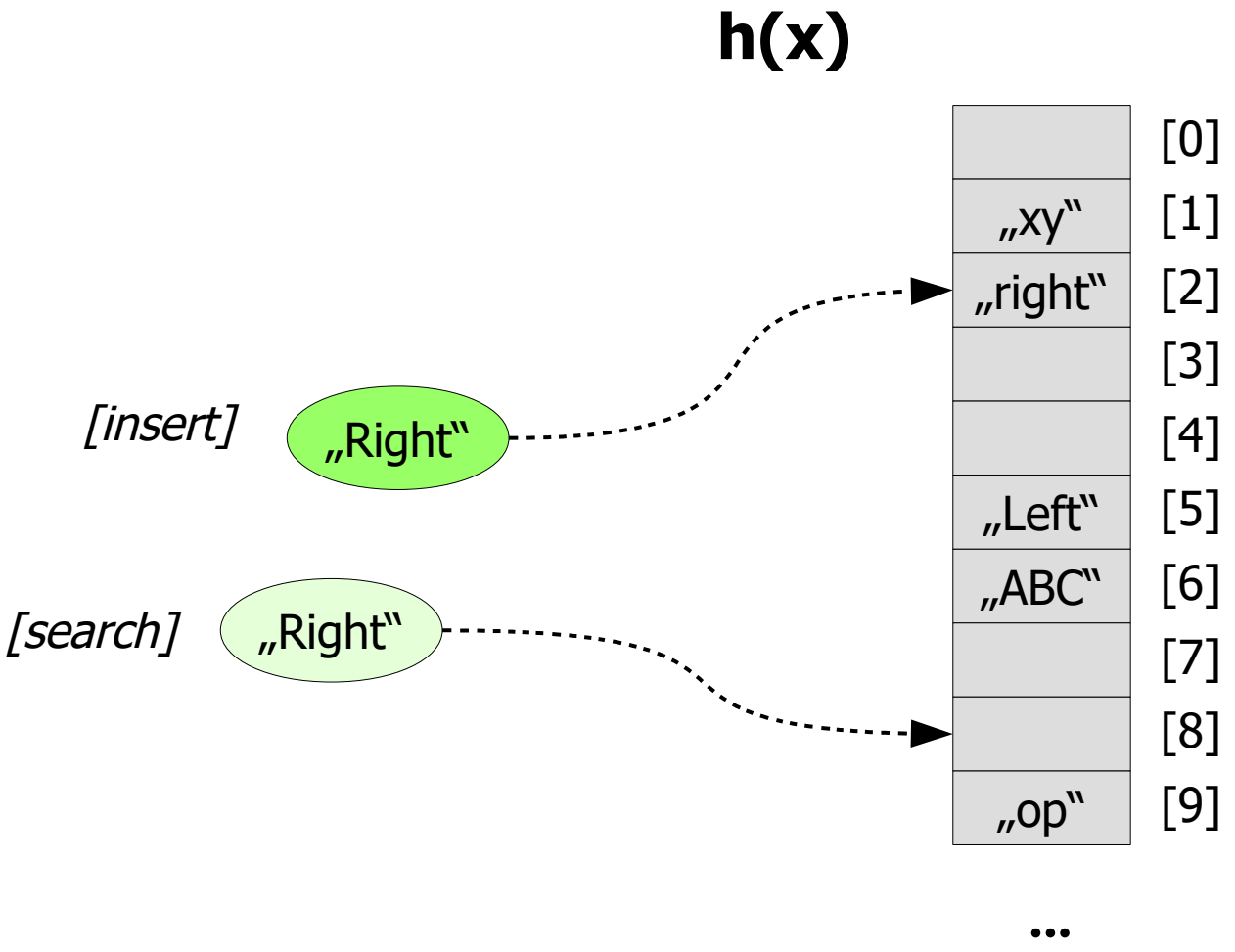
Note, the opposite can never happen:



... because the hash function must always work in the same way.

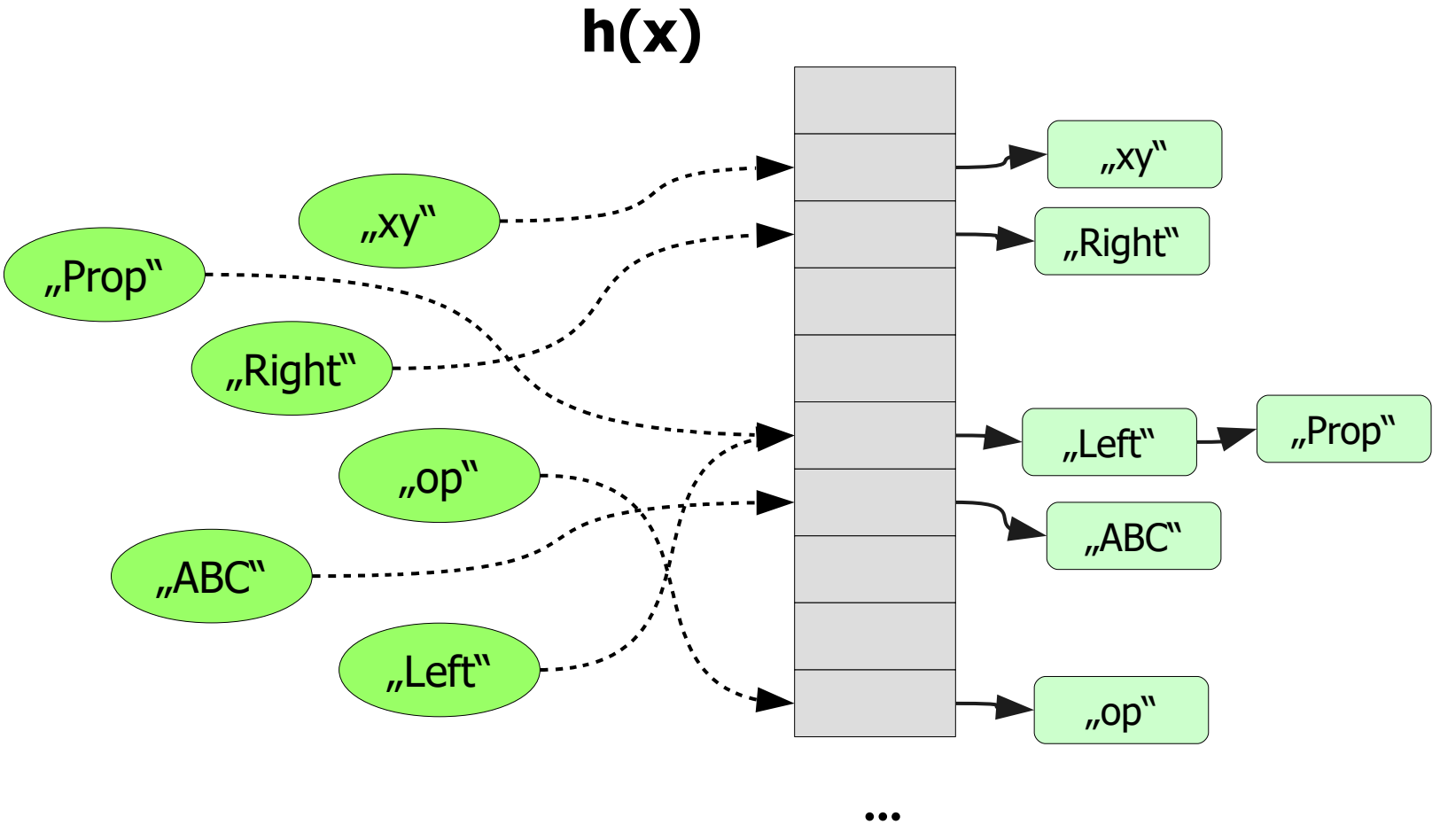
Collisions

Otherwise we might add an object in one cell, and later search for it in another cell:



Collisions

The simplest way of resolving collisions is through linked lists:

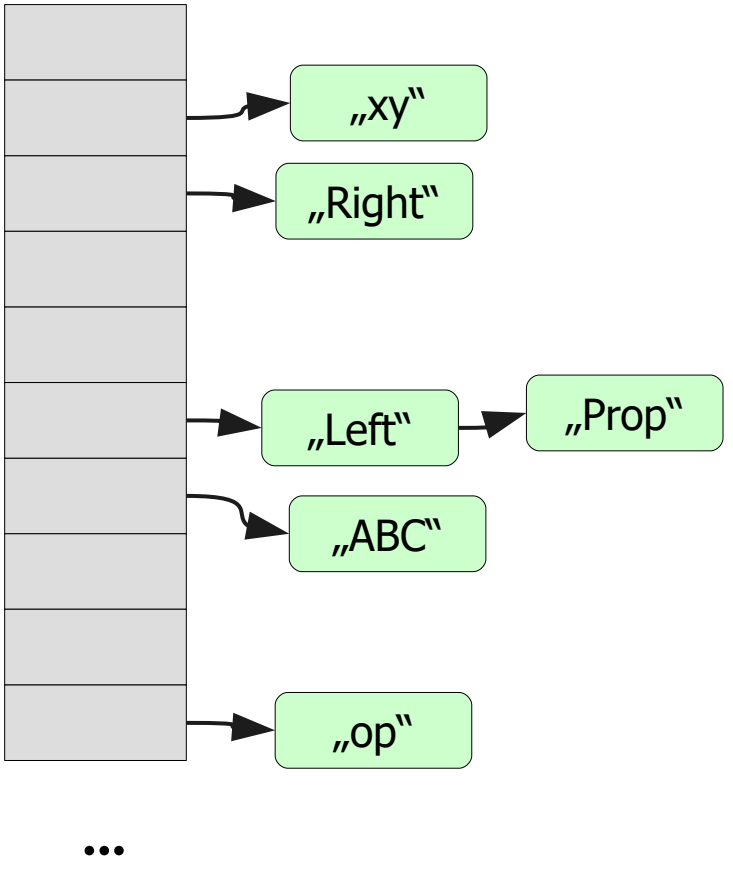


Collisions

The simplest way of resolving collisions is through linked lists.

So our table is no longer array of values, but...

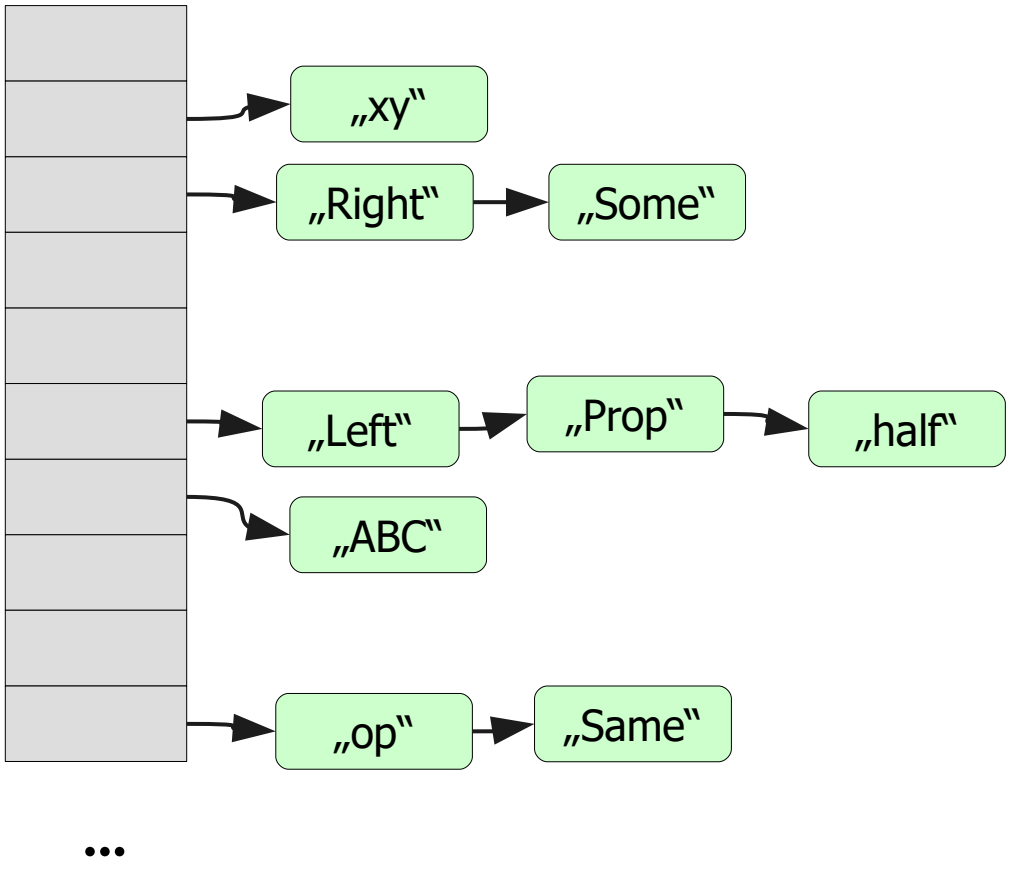
... array of linked lists of values.



Collisions

Note, over time the lists can become longer,

... though, we will try to keep them all short.



Collisions

Which results in the following time complexities:

Operation	*Time complexity*
Insert	**O(1)**
Search	average **O(1)**
Remove	average **O(1)**

Question: Why the time complexity for insertion differs?

Collisions

Which results in the following time complexities:

Operation	*Time complexity*
Insert	**O(1)**
Search	average **O(1)**
Remove	average **O(1)**

Question: Why the time complexity for insertion differs?

Answer: Because we can always add new item to front of a linked list, while searching and removing an item require scan over the list.

Exercise

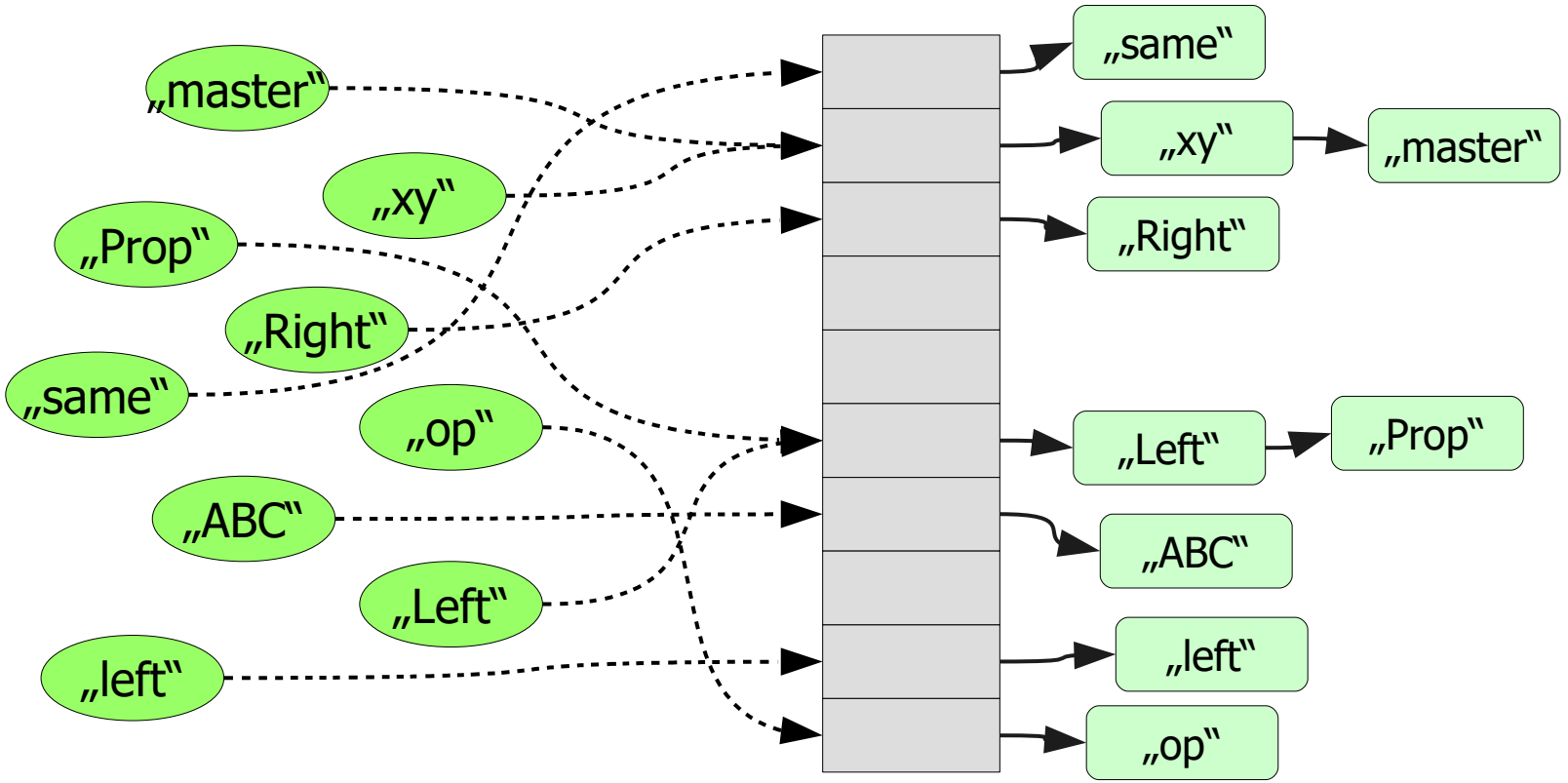
Implement the described scheme,

- store strings,
- use some trivial hash function.

Good vs. bad hash functions

The key important fact for any hash table is to have a good hash function.

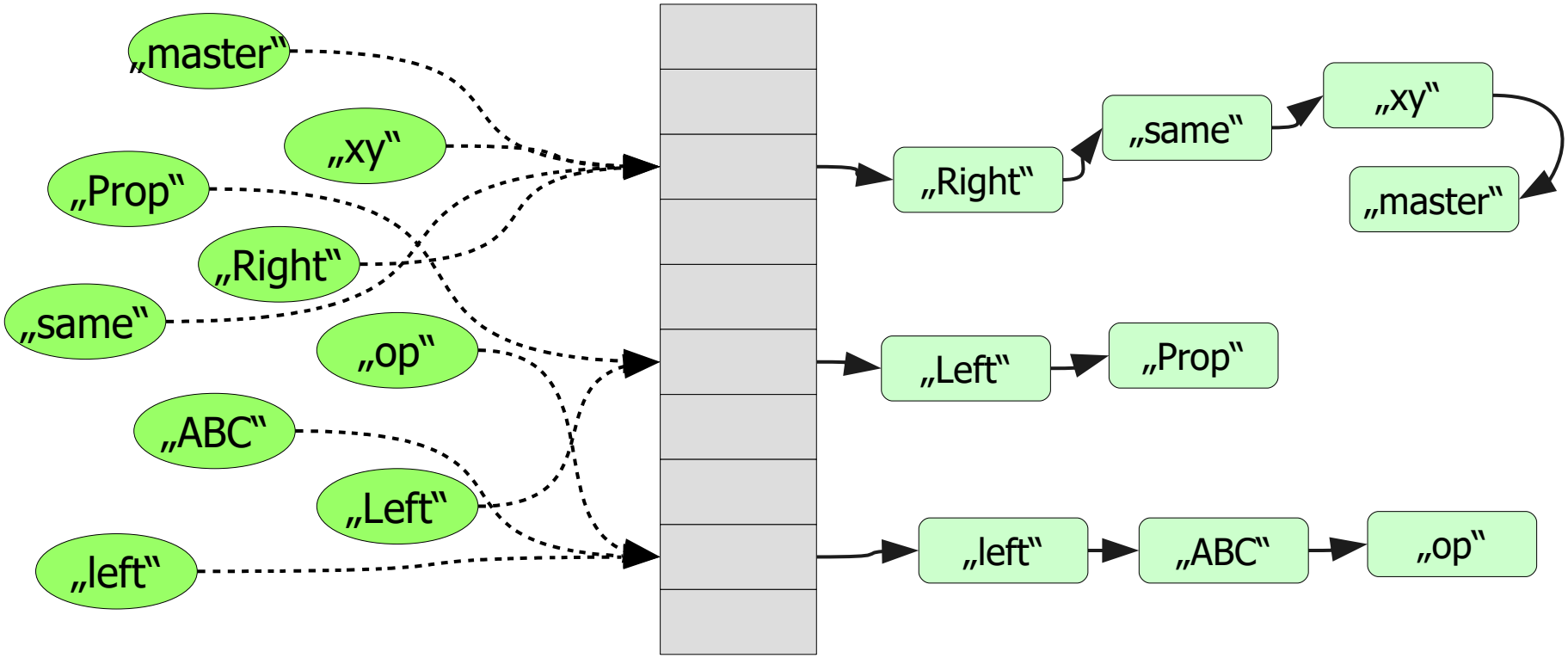
Here by "good" we mean that it spreads values good enough:



Good vs. bad hash functions

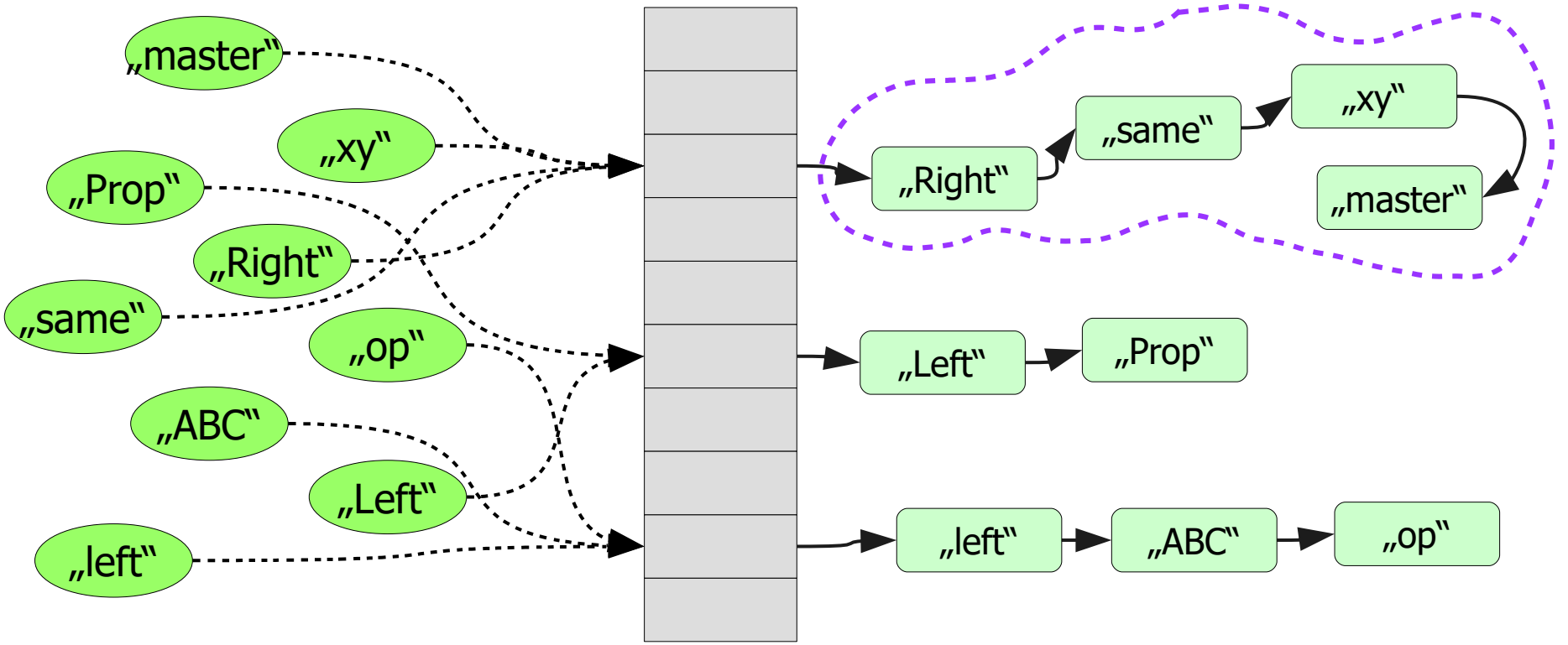
In contrast to it, "bad" hash function will spread the values badly,

... creating lots of collisions:



Good vs. bad hash functions

If many objects fall into the same cell, performance of hash table degrades to performance of singly linked list.

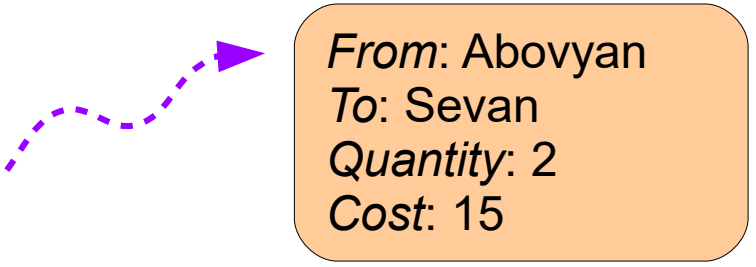


Good vs. bad hash functions

There are several properties, that a good hash functin must satisfy:

1) It must consider all logical parts of the object:

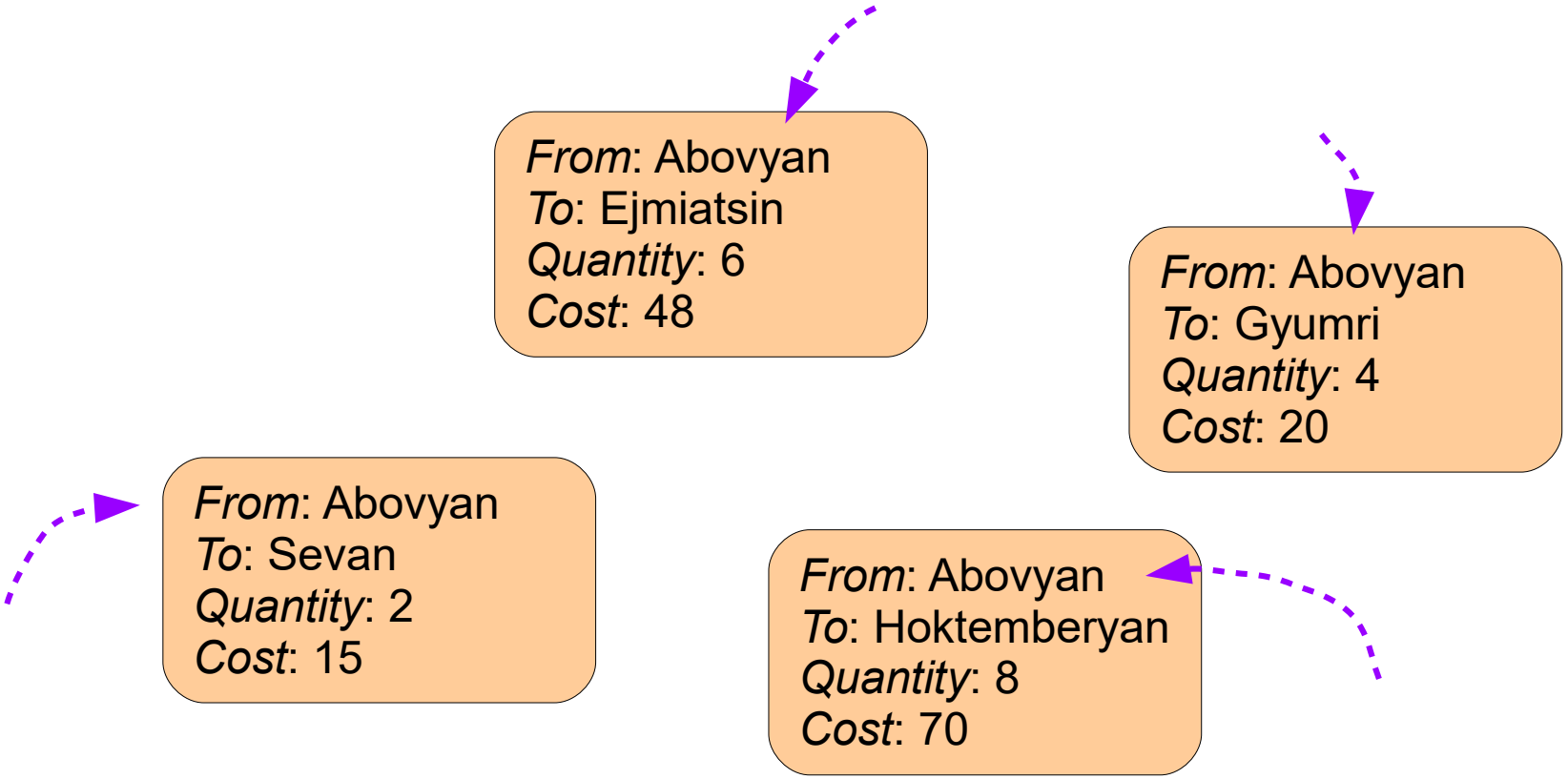
What can happen <u>if hashing only some part</u> of an object (DB entry)?



Good vs. bad hash functions

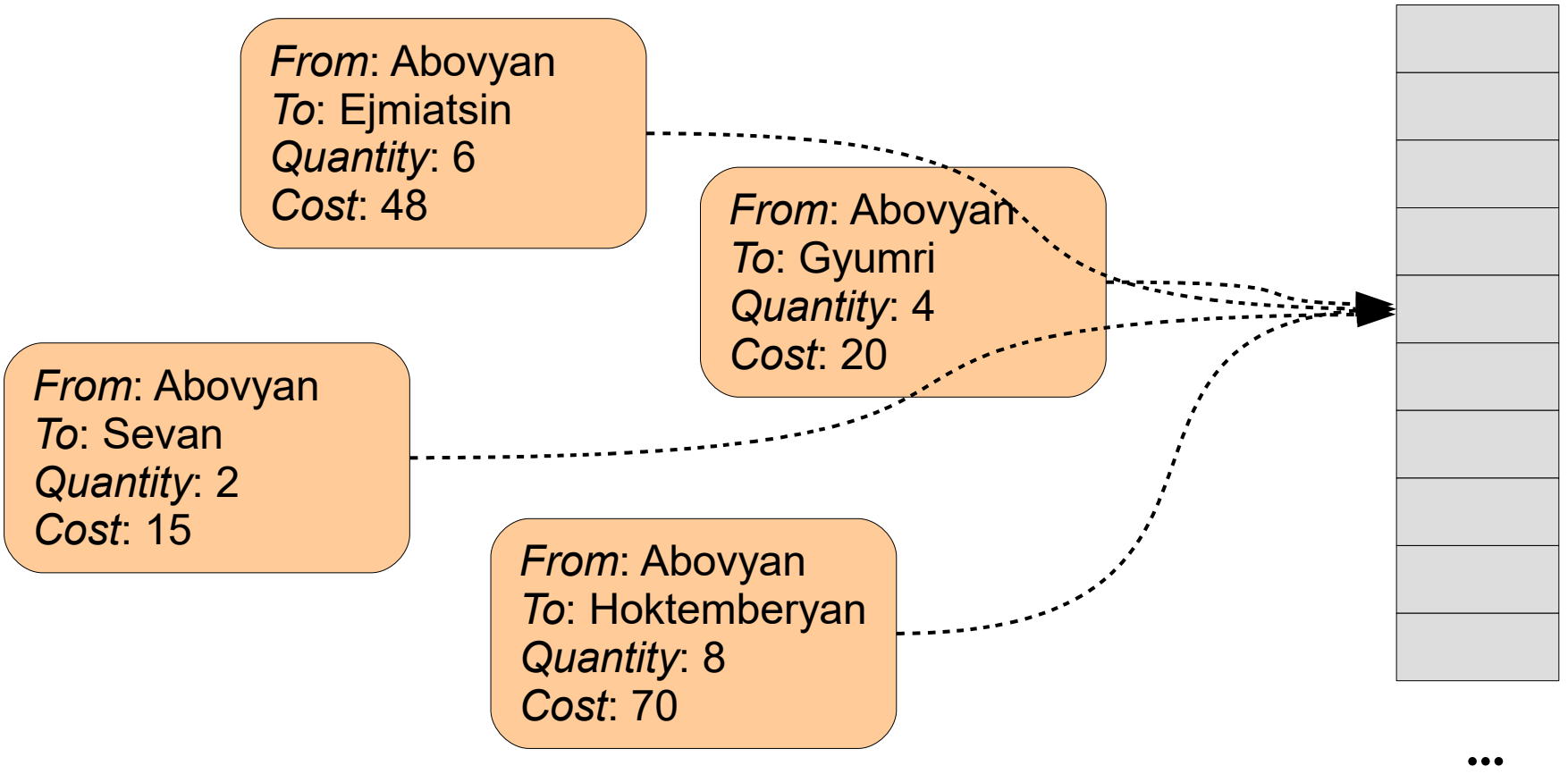
Maybe in current problem, there is a factory located in "Abovyan", so most of items will have:

```
<from> == "Abovyan"
```



Good vs. bad hash functions

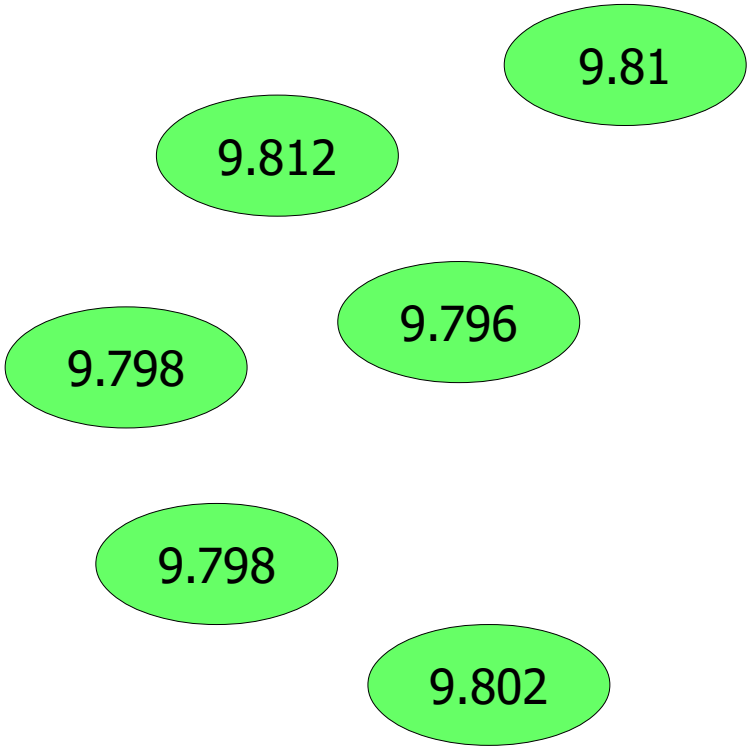
So they all will go into the same cell of hash table.



Good vs. bad hash functions

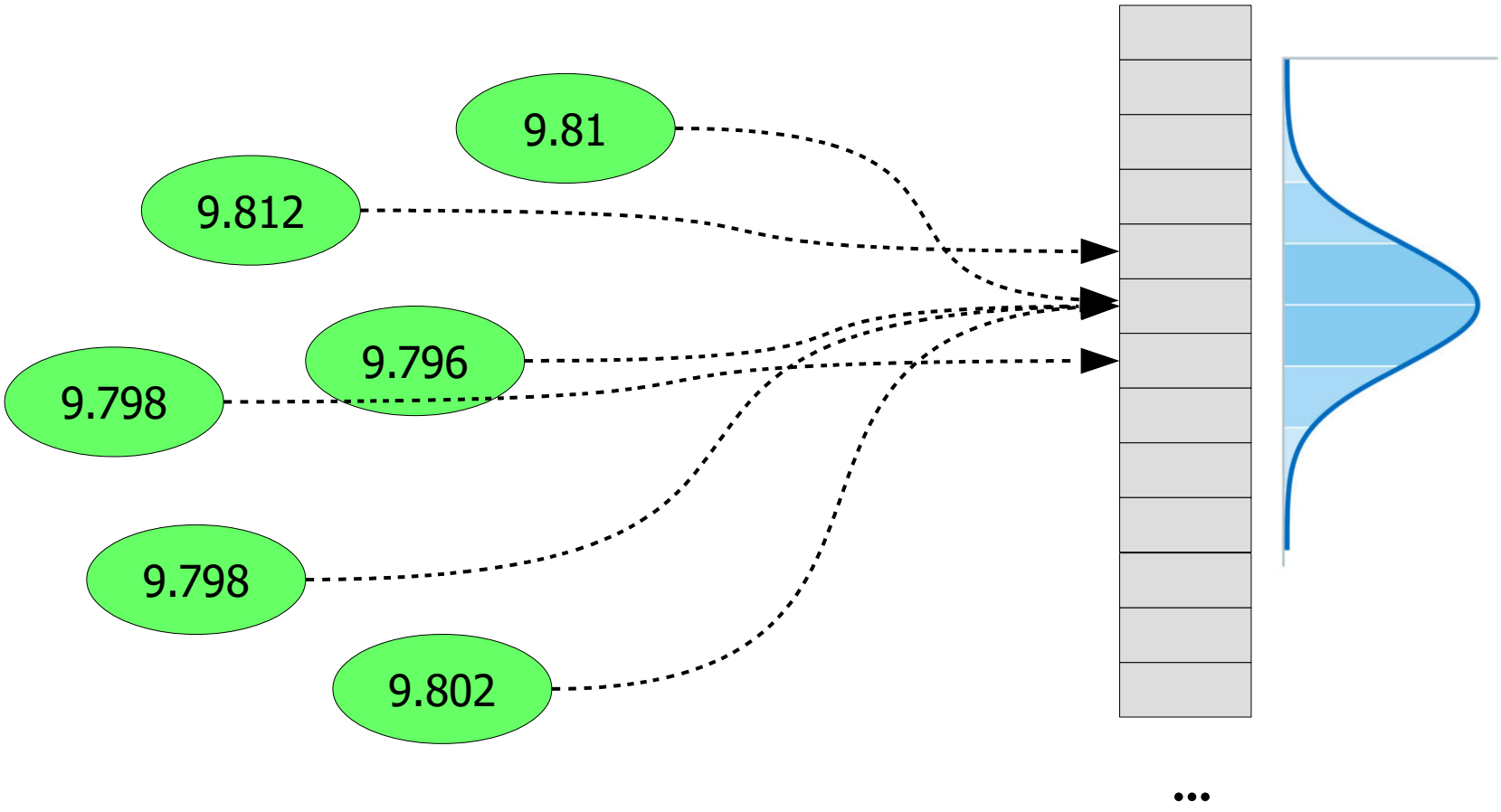
2) For similar objects it must not return similar values.

What if we are measuring the constant "**g≈9.8**"



Good vs. bad hash functions

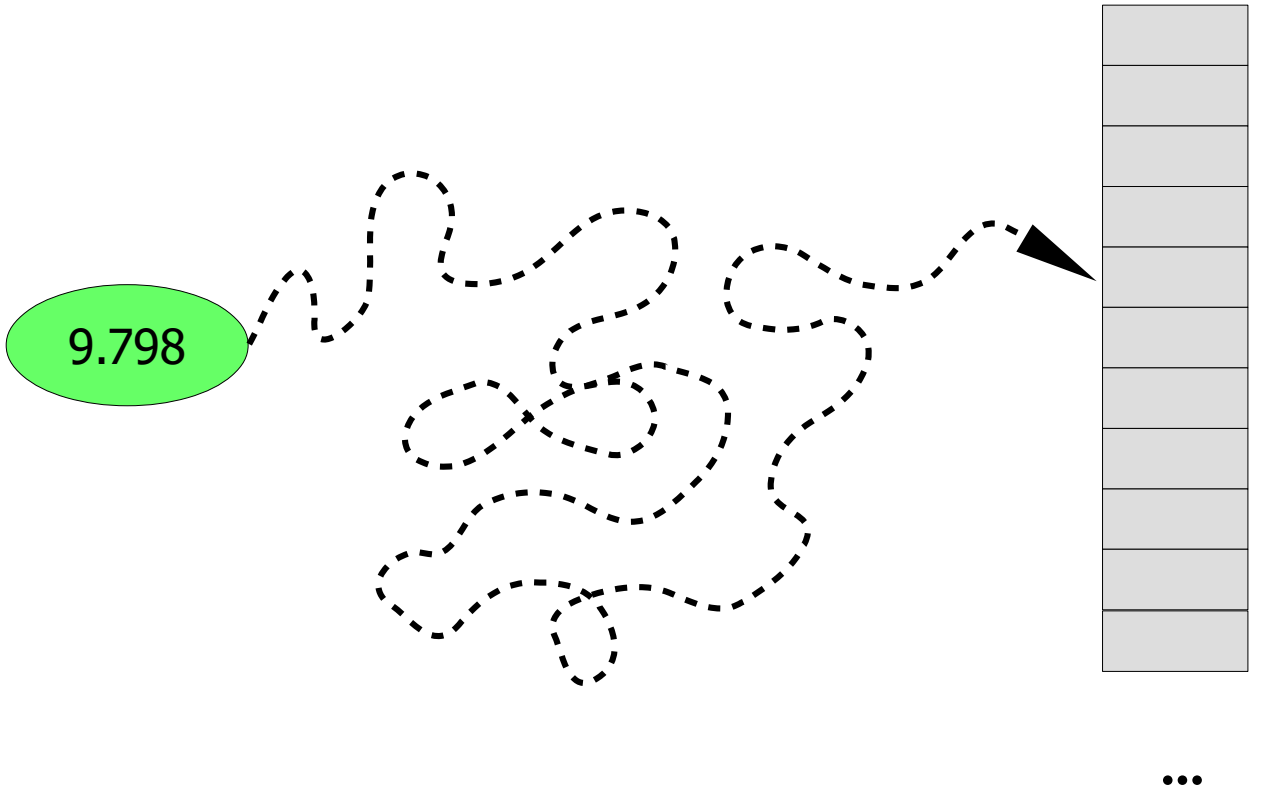
Violating this rule will place them all in few adjacent cells of the table:



Good vs. bad hash functions

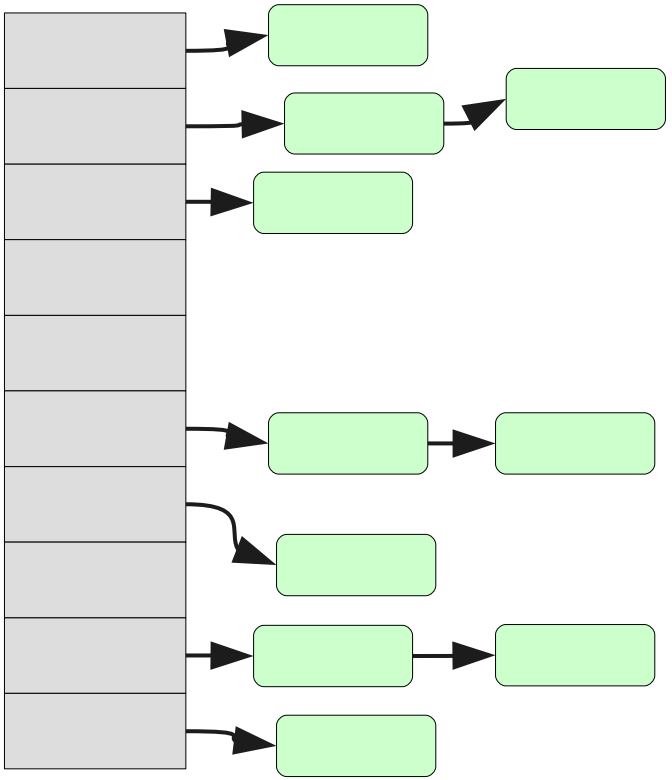
3) And finally, good hash function should work fast enough,

... as any operation on hash table involves hash value calculation.



Good vs. bad hash functions

Question: Why in hash tables we use linked lists, and not more performant structures like BSTs?

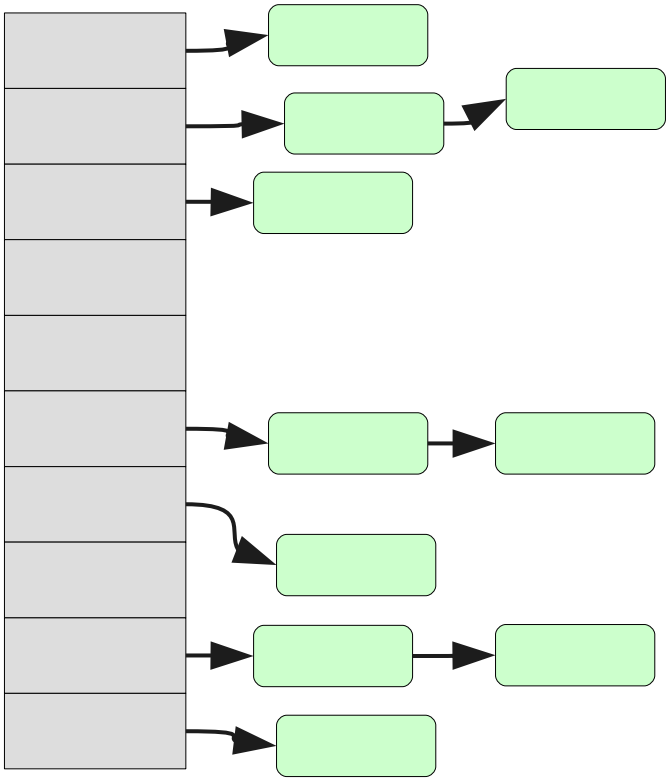


Good vs. bad hash functions

Question: Why in hash tables we use linked lists, and not more performant structures like BSTs?

Answer: Because good hash table always keeps those lists short.

If lists are getting longer, we must reorganize something else, and not optimizing their way of storage.



Writing good hash functions

<content from "hash functions">

Exercises

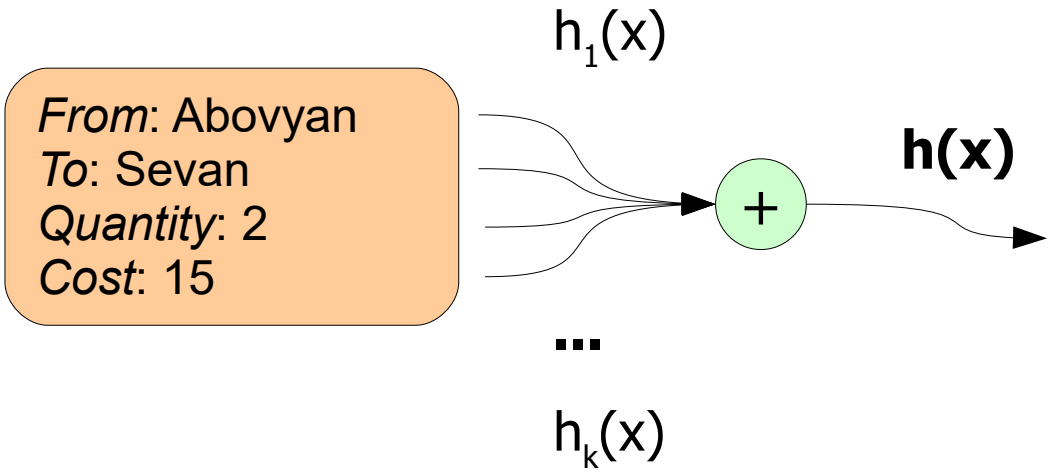
1) Write some good and bad hash functions for 3D points (integer coordinates).

2) For both scenarios:

- **2.1)** Fill lot of points in a hash table,
- **2.2)** Compare how they are distributed.

Writing good hash functions

Sometimes we can hear - "good hash function for an object is just sum of hashes of all its fields".



This can be true in many cases, but not always:

Writing good hash functions

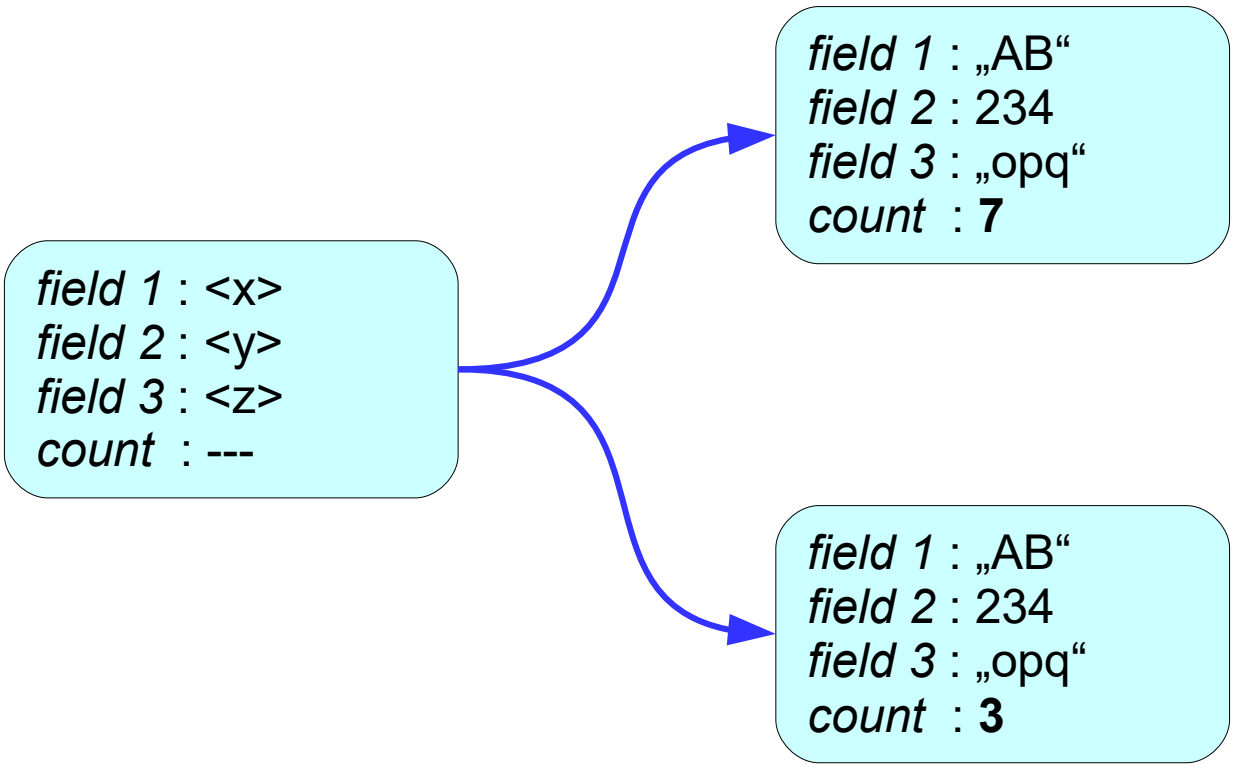
Case 1) What if the object has some counter...

... of how many times it was modified.

field 1 : <x>
field 2 : <y>
field 3 : <z>
count : 7

Writing good hash functions

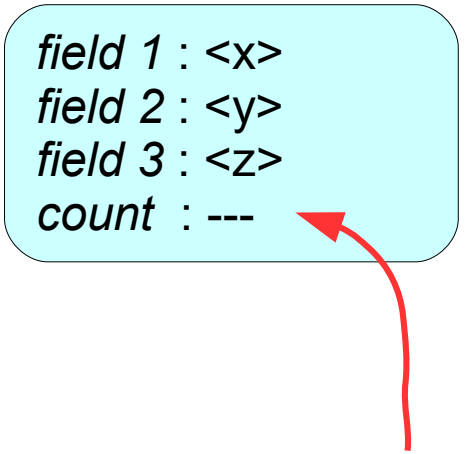
It can be brought to the same state by different numbers of modifications:



... and those **2** objects are logically the same.

Writing good hash functions

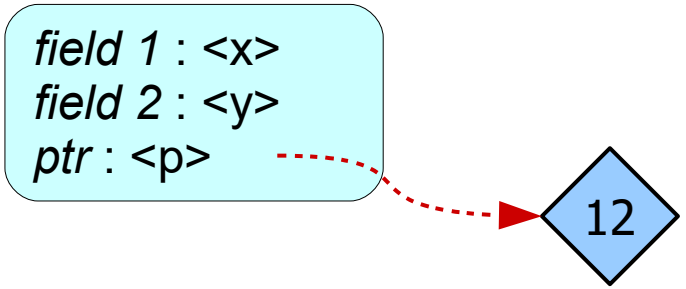
So "counter" should just not participate in the hash.



Writing good hash functions

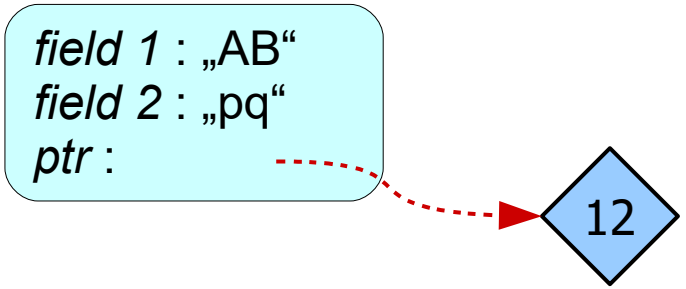
Case 2) What if the object refers to some other object (on a heap)...

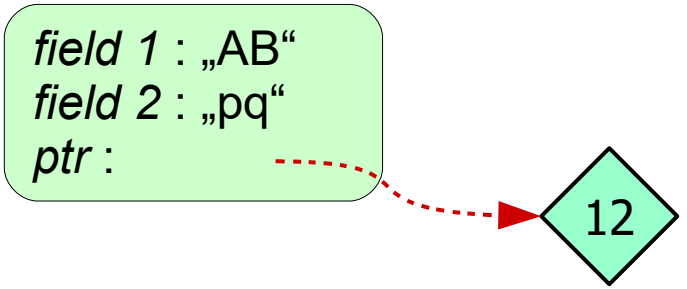
... should we just consider address of the pointer into the hash?



Writing good hash functions

No, because two objects might point to **2** helper objects, which are logically the same.





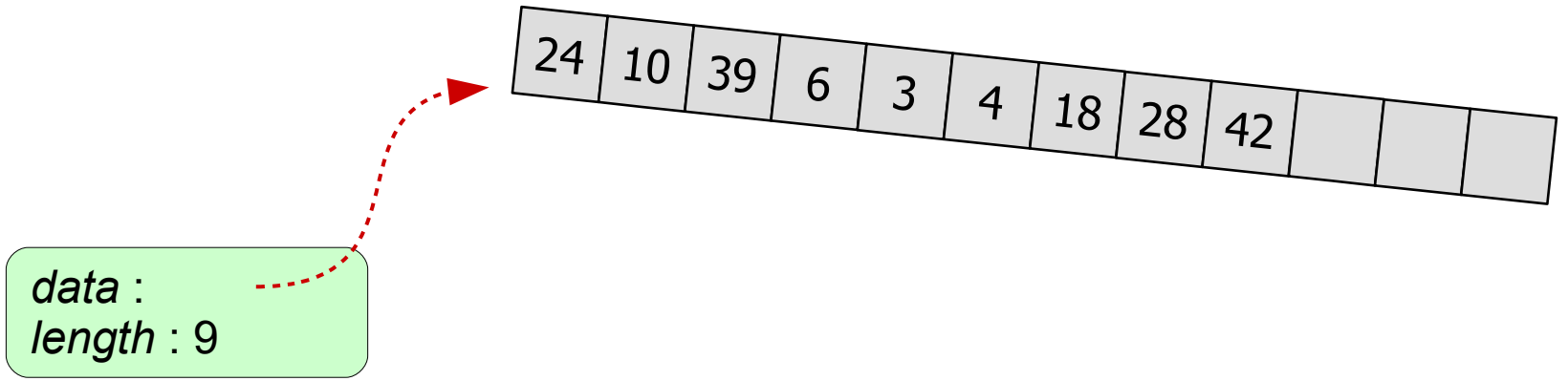
So they <u>should be considered as the same</u>,

... while adding pointers into the hash will result in different hash value.

Writing good hash functions

A well-known example is the dynamic array,

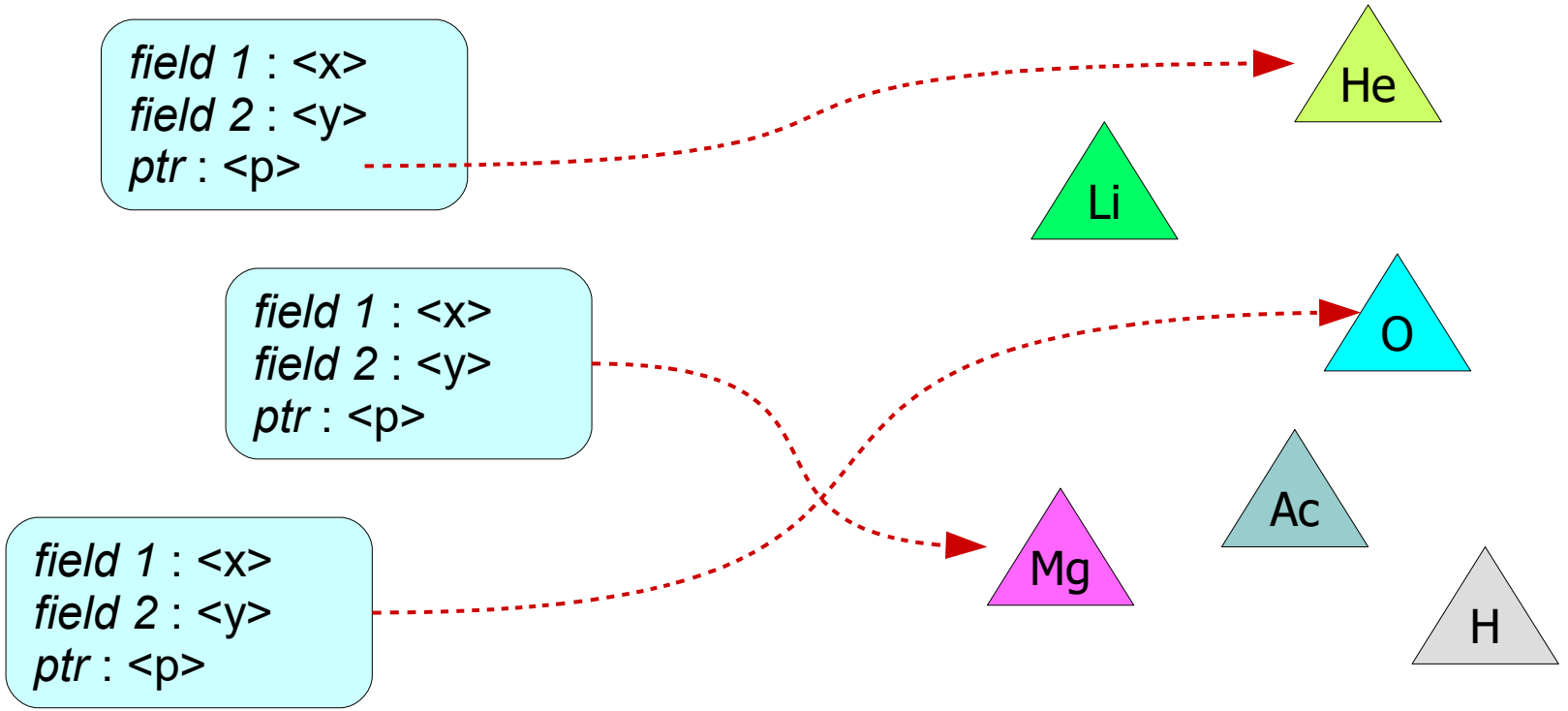
... where we should compare the actual data on the heap.



Writing good hash functions

Case 3) There can be cases when we have a collection of unique helper objects,

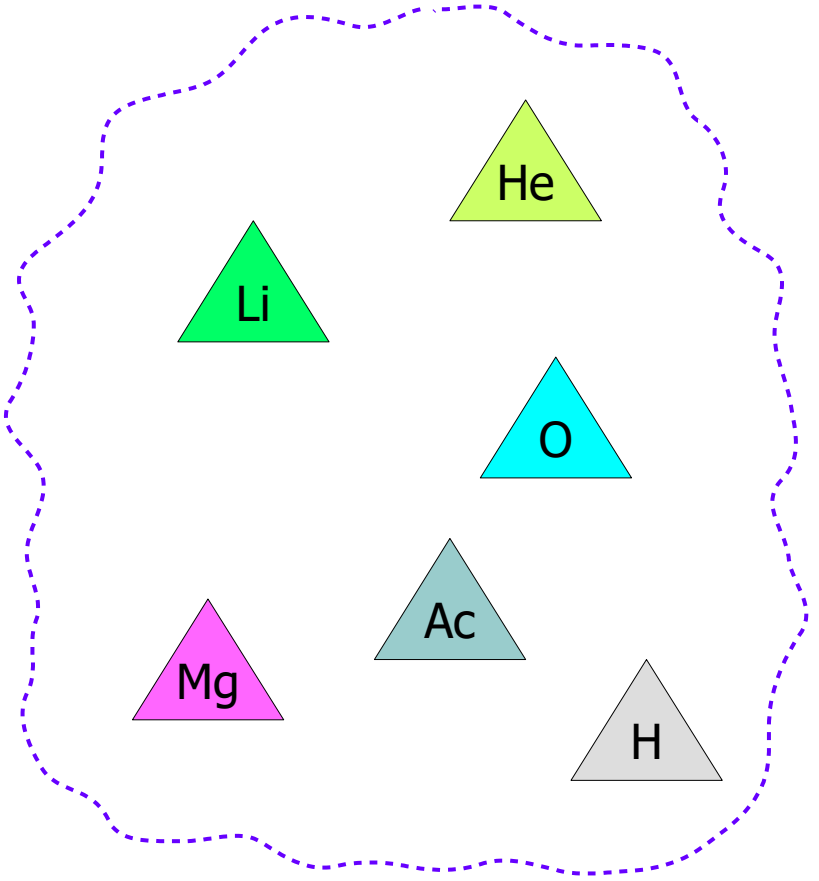
... while other objects do refer to them.



Writing good hash functions

The helper objects are unique, so it makes no sense to compare them,

... instead, we can just compare the pointers.



Writing good hash functions

As a conclusion:

- There is no universal formula for writing good hash function.
- We should do it by looking into details of current problem.

Load factor

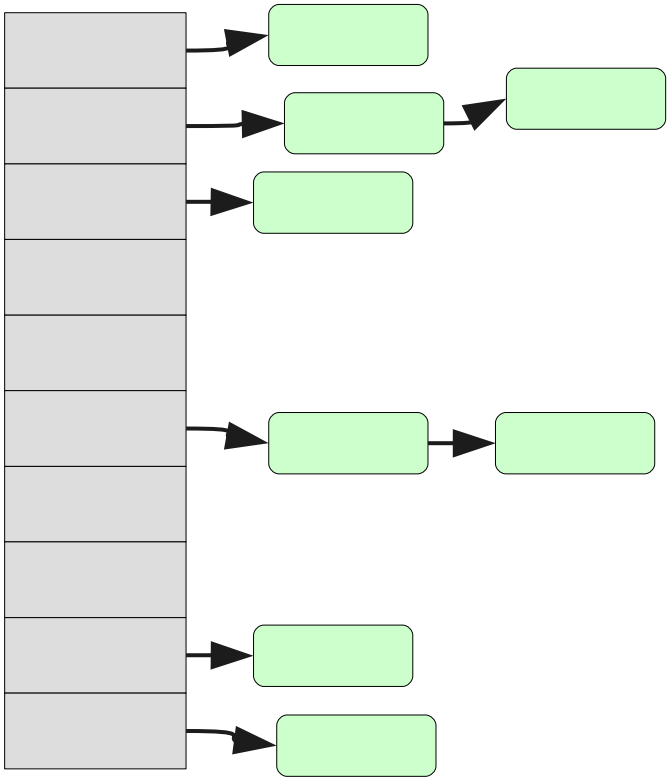
Load factor is the primary statistics of any hash table.

$$f = \frac{N}{M}$$

In the presented example:

- **N = 8**,
- **M = 10**,
- **f = 0.8**

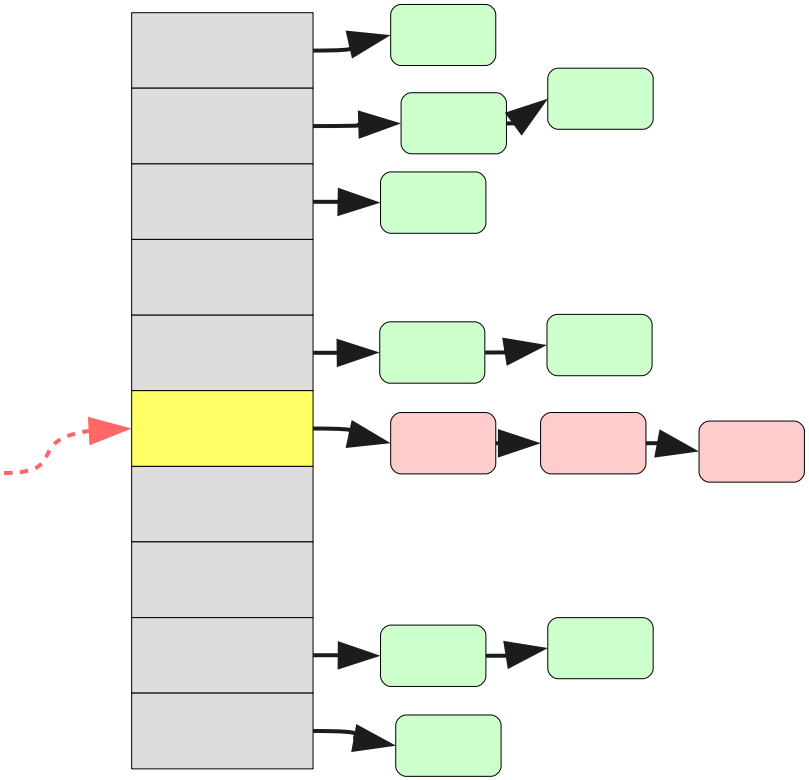
In case of closed addressing "**f**" corresponds to average length of the linked lists.



Load factor

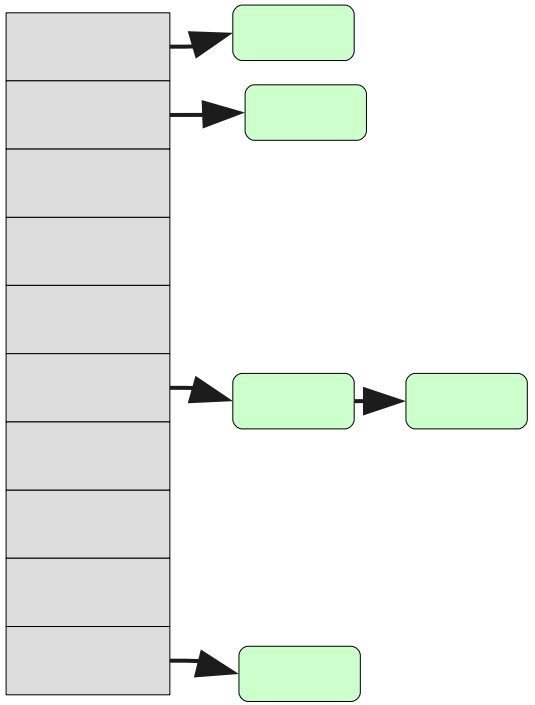
This means that "**f**" also corresponds to the average time required to search / remove in the hash table,

... as those operations require scan of corresponding list.

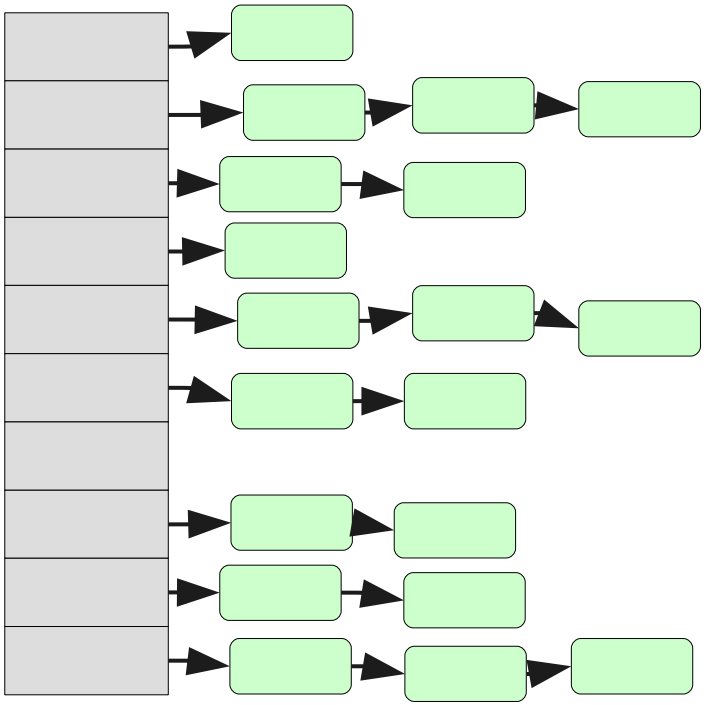


Load factor

That's why in order to keep hash table efficient, we should follow for the factor to remain in a certain range, like in **[0.5-2.0]**.



f = 0.5



f = 2.0

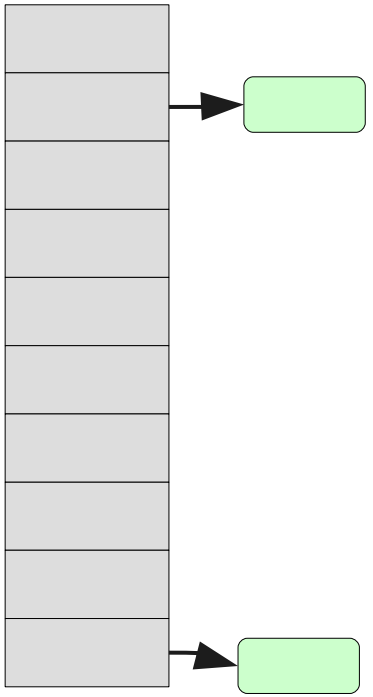
Load factor

In STL library we have the following method for knowing load factor:

```
float std::unordered_set<>::load_factor() const
          unordered_multiset<>
          unordered_map<>
          unordered_multimap<>
```

Load factor

Question: What will happen if the load factor will be too low?

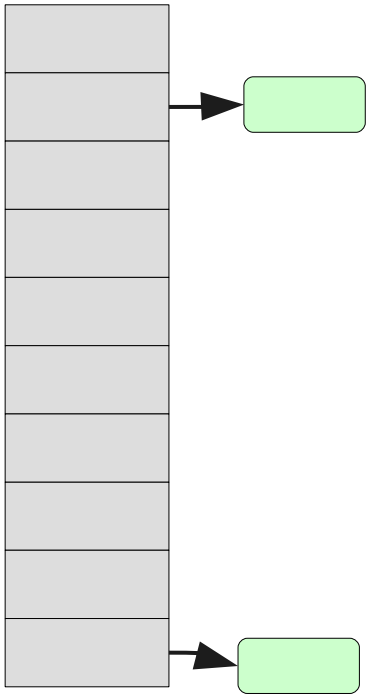


f = 0.2

Load factor

Question: What will happen if the load factor will be too low?

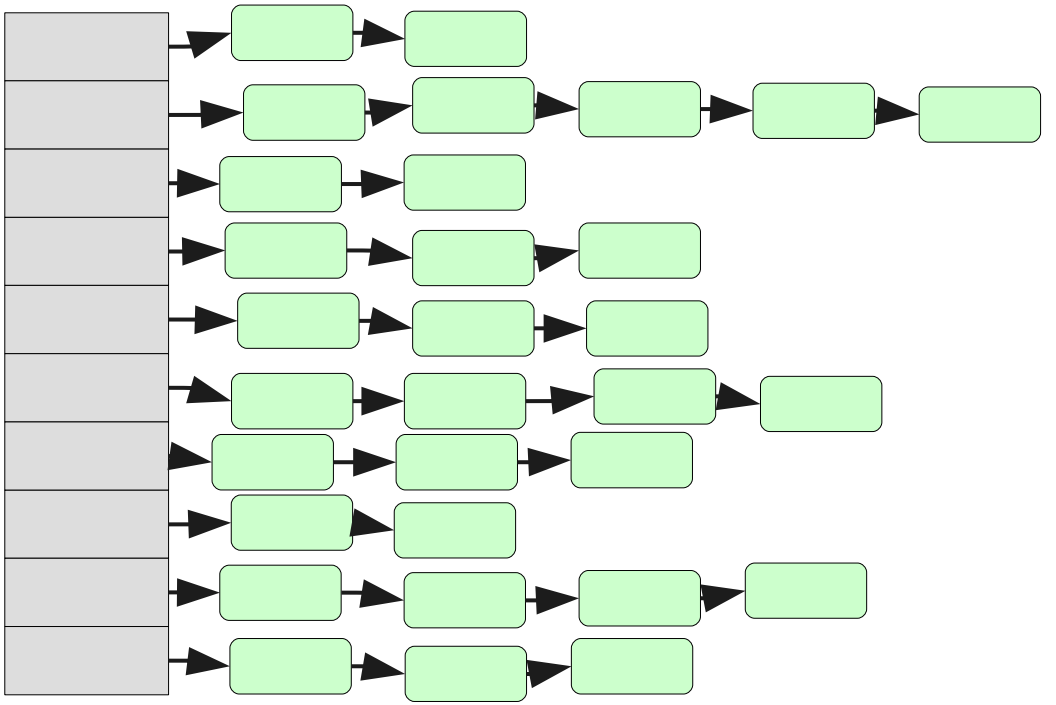
Answer: This will result in a waste of memory,
when having lots of empty linked lists.



f = 0.2

Rehashing

Regardless of how good the hash function is (how evenly it distributes values), there will come time when linked lists will become too long.

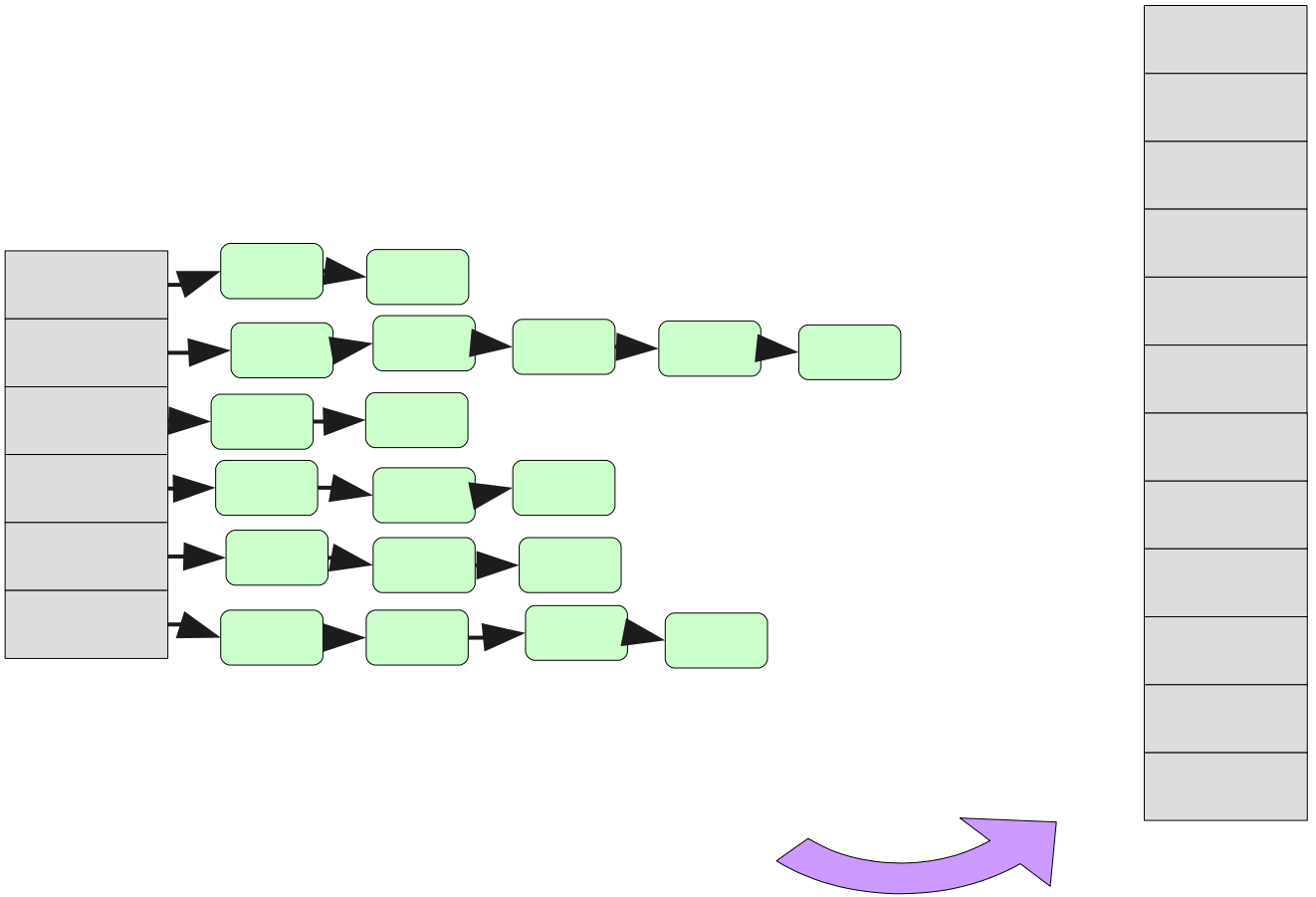


f = 3.1

Rehashing

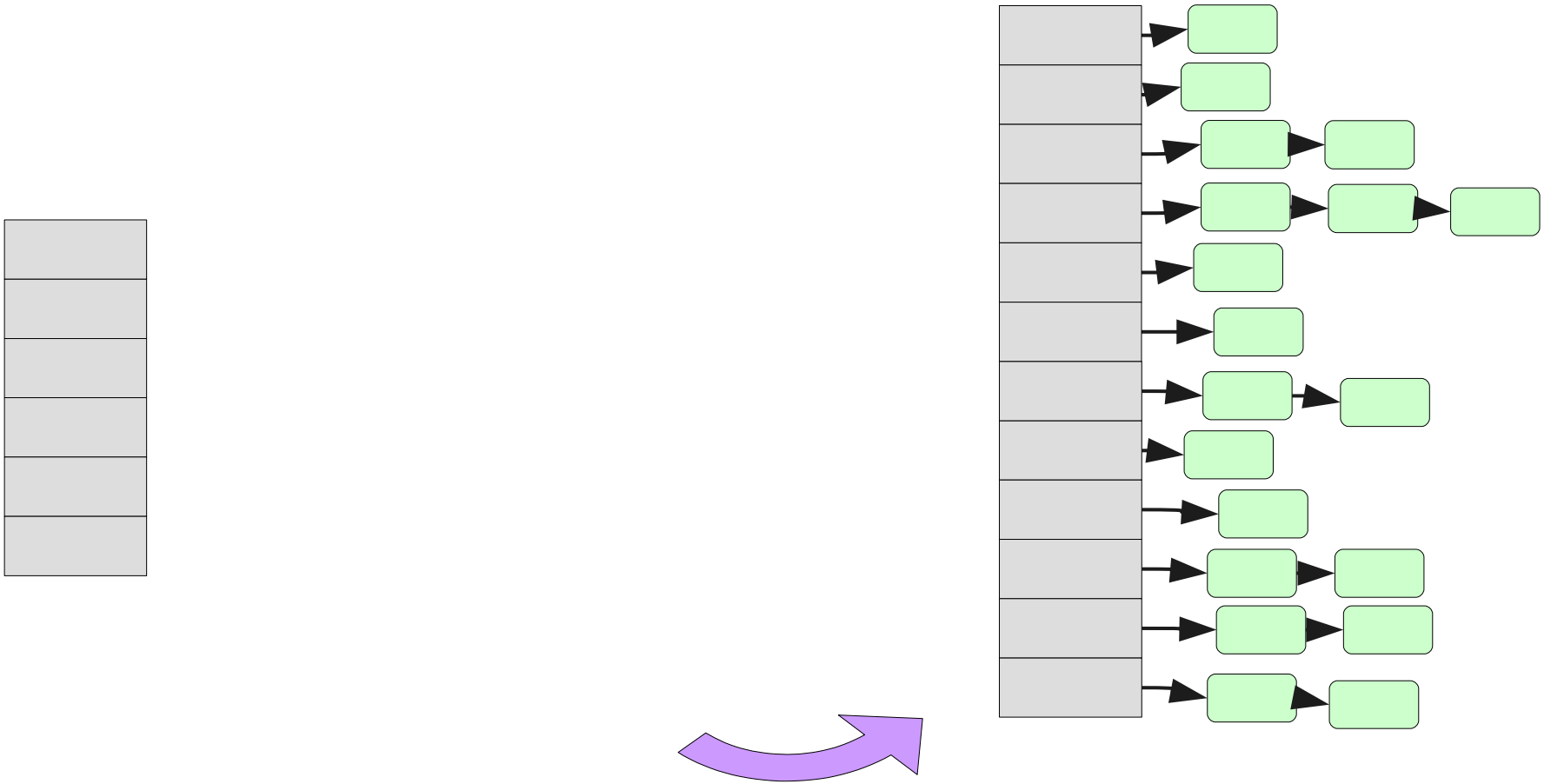
In order to keep load factor in desired range, we:

1) Allocate a larger array of slots:



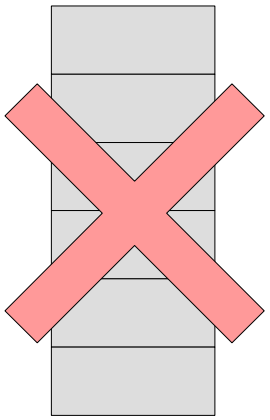
Rehashing

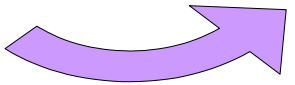
2) Copy all existing values in their new positions:

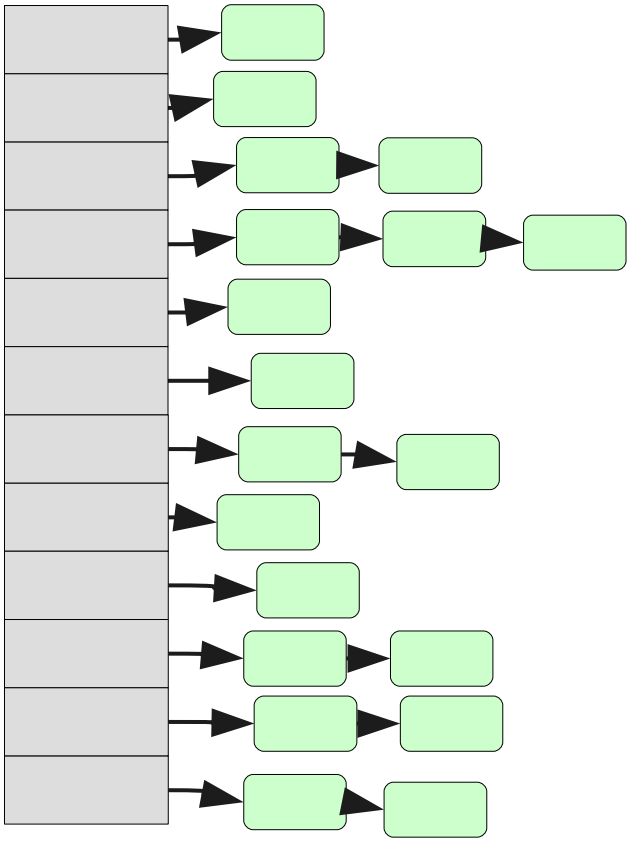


Rehashing

3) Deallocate the old array.







Rehashing

This operation is called "rehash".

Let's point that its time complexity is **O(N)**.

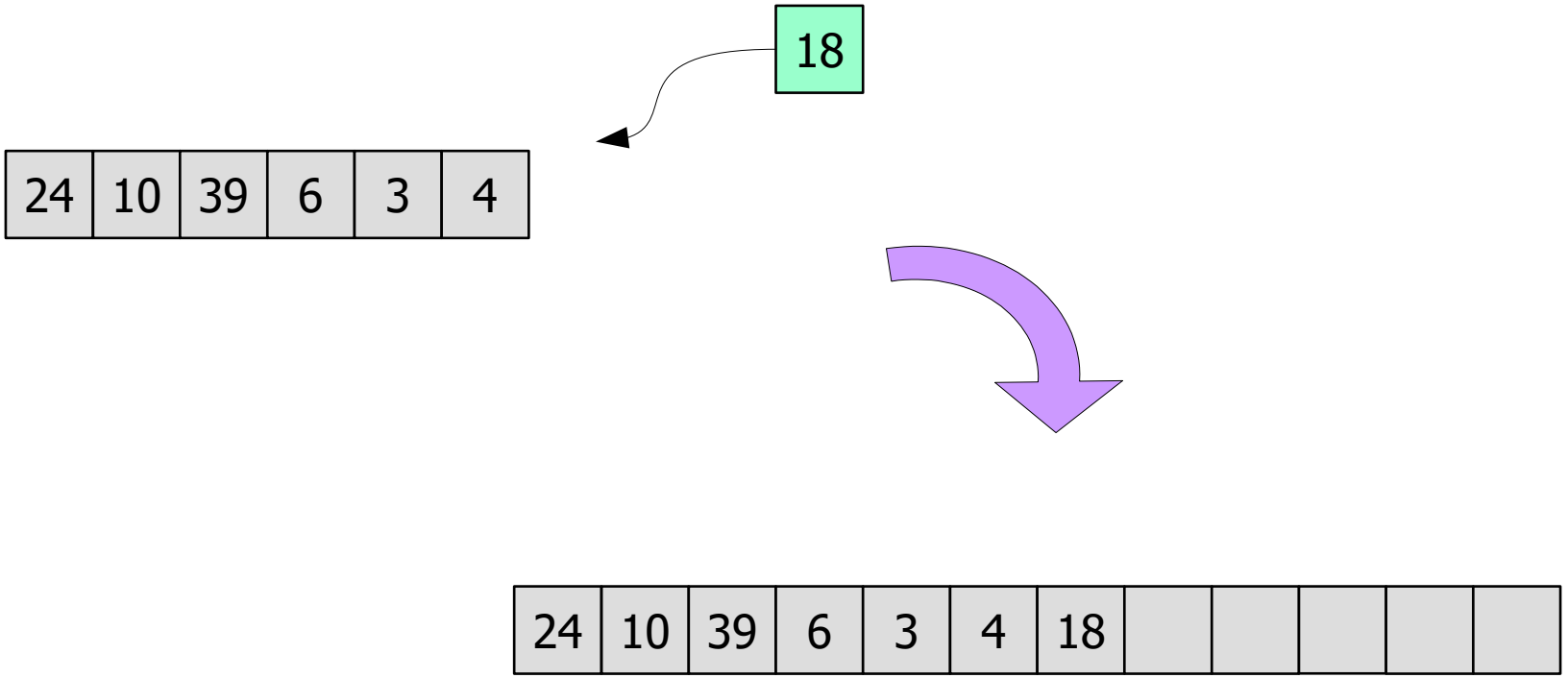
This means that time complexity of insertion can be up to **O(N)**.

... however at average it is **O(1)**,

... this means that time complexity of insertion is amortized **O(1)**.

Rehashing

In fact it becomes quite similar to reallocation in dynamic array.



Rehashing

Methods which deal with hash policy:

```
float max_load_factor() const;
void max_load_factor ( float z );
void rehash ( size_type n );
```

Exercises

2) Implement rehash, for both cases:

* when load factor becomes greater than upper threshold,
* when it becomes less than lower threshold.

Lazy rehash

The structure which we describe provides **O(1)** amortized time for all operations,

... which is good for most of practical cases,

... and which is actually implemented in STL or other standard libraries.

`std::unordered_set<>`

`std::unordered_multiset<>`

`std::unordered_map<>`

`std::unordered_multimap<>`

Lazy rehash

However, it is not always OK.

- Imagine a server, which operates on a hash table,
- It does ordinary insert / search / remove,
- For a new request it adds an entry in the hash table,

Lazy rehash

However, it is not always OK.

- Imagine a server, which operates on a hash table,
- It does ordinary insert / search / remove,
- For a new request it adds an entry in the hash table,
- And some insertion caused a rehash:
  - so data must be copied on the HDD, in **O(N)** time,
  - which might take **1-2** minutes,
  - and all user must wait...

This is, obviously, not OK.

Lazy rehash

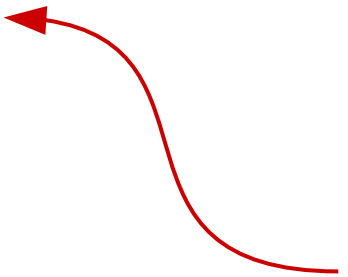
What should we do then? Can't we use hash tables on servers?

Lazy rehash

The idea behind lazy rehash is to achieve same result (a rehash), but not instantly.

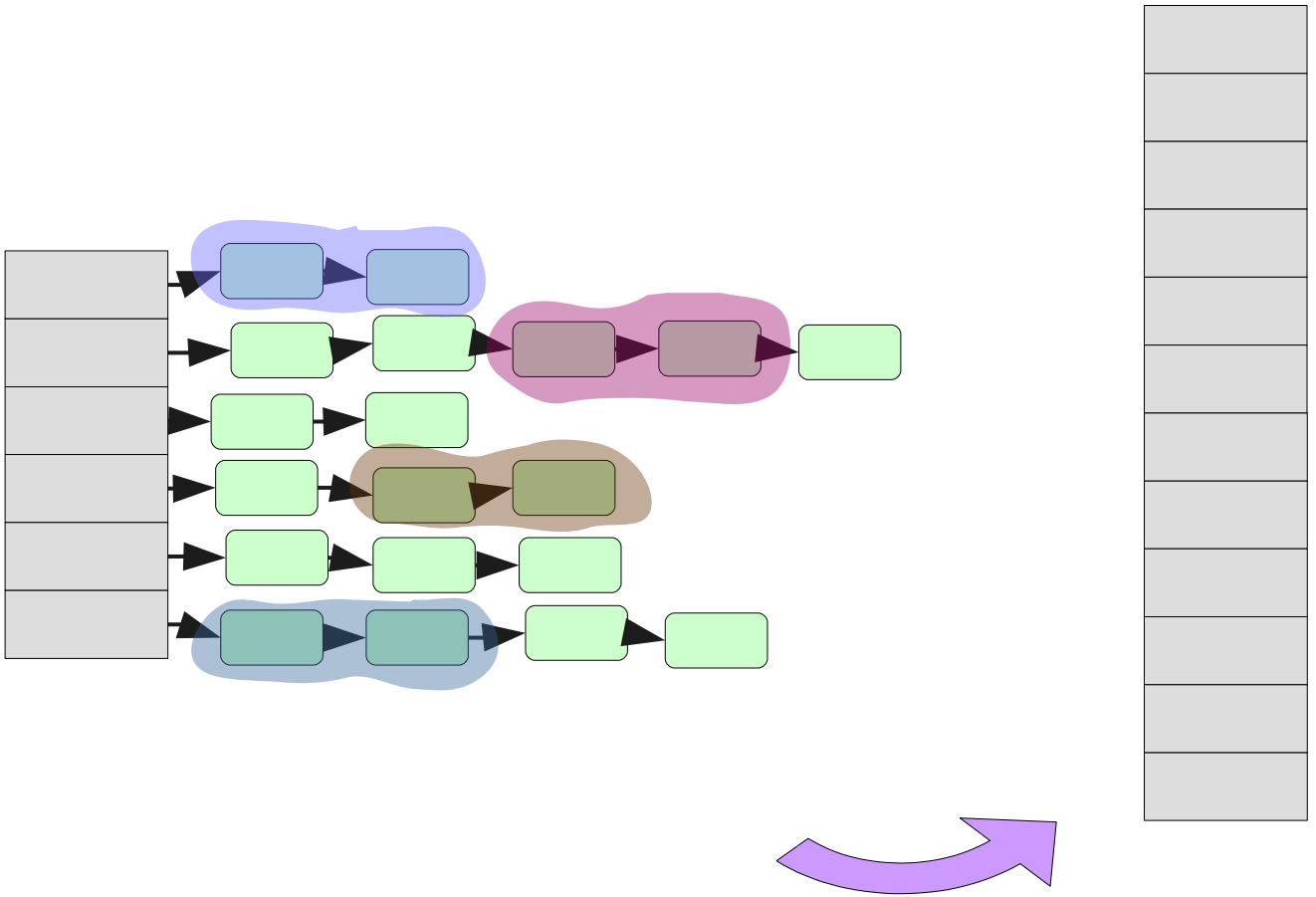
We have **3** phases during a rehash, but the significant time goes only on the second one:

- allocate new array,
- copy values there from old array,
- deallocate old array.



Lazy rehash

So instead of moving it at once, we will move it by small portions:

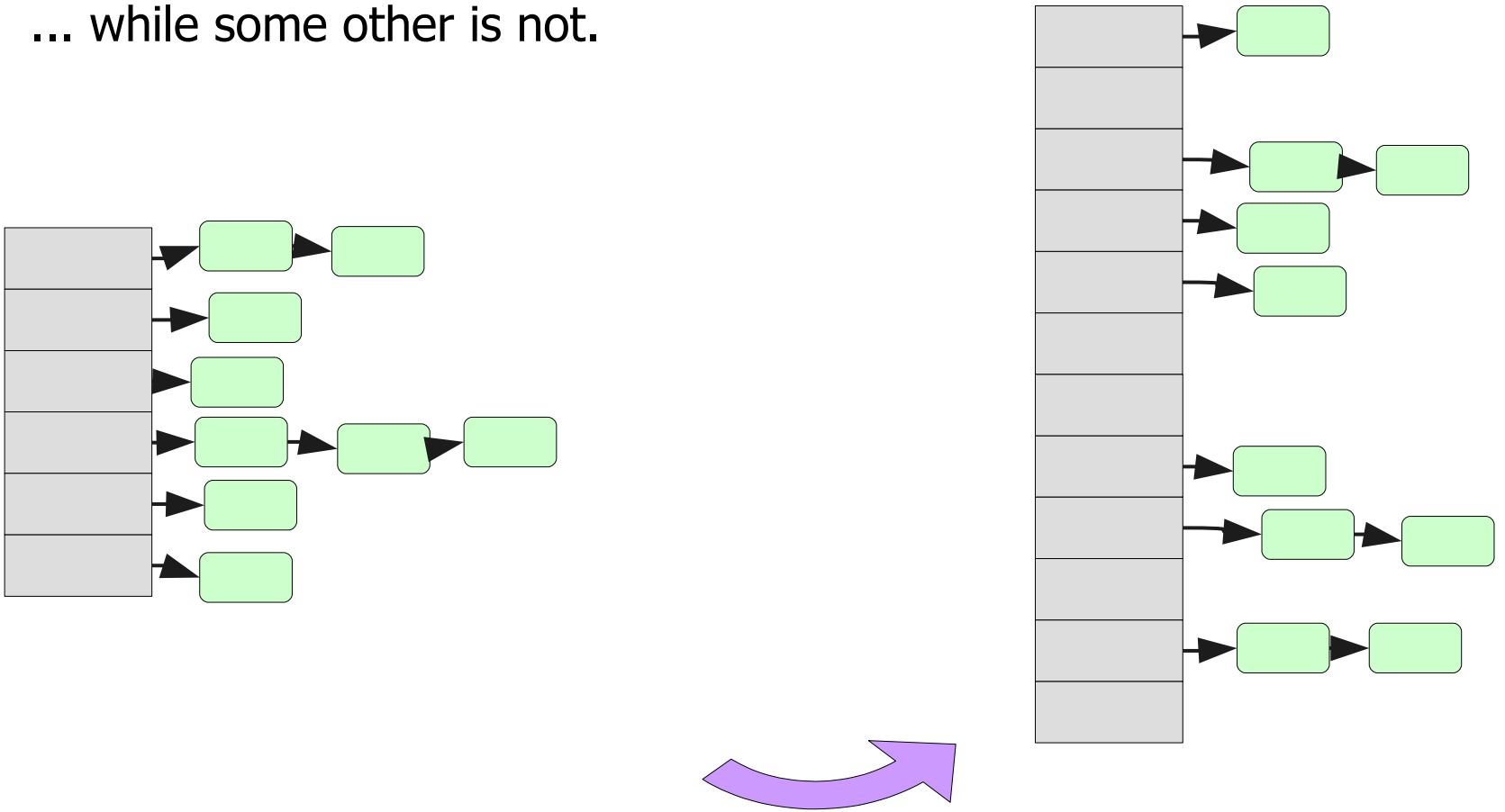


Lazy rehash

This means that our hash table should remain operable also:

... when some portion of data is moved,

... while some other is not.



Lazy rehash

Codes of the main operations do change then:

```
function search( x: Key ) : Value
    if <in intermediate state>
        // Search in old table
        ...
        // If not found, search in new table
        ...
    else
        // Search in the only table
        ...
```

Lazy rehash

Codes of the main operations do change then:

```
procedure remove( x: Key )
    if <in intermediate state>
        // Remove from old table
        ...
        // If not found, remove from new table
        ...
    else
        // Remove from the only table
        ...
```

Lazy rehash

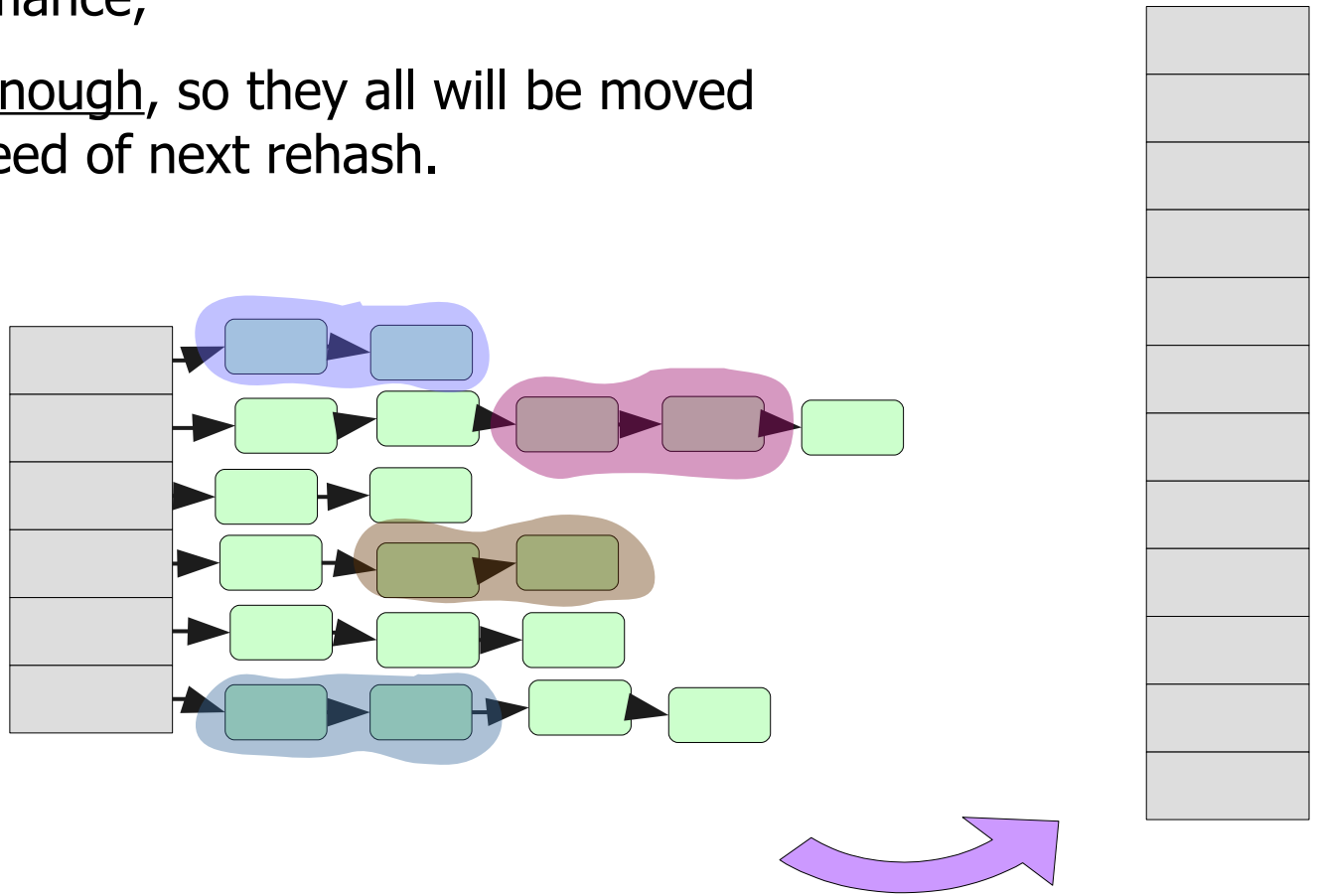
Codes of the main operations do change then:

```
procedure insert( x: Key, y: Value )
    if <in intermediate state>
        // Insert right in the new table
        ...
    else
        // Insert in the only table
        ...
```

Lazy rehash

The moved portions must be:

- small enough, to not affect the performance,
- large enough, so they all will be moved until need of next rehash.

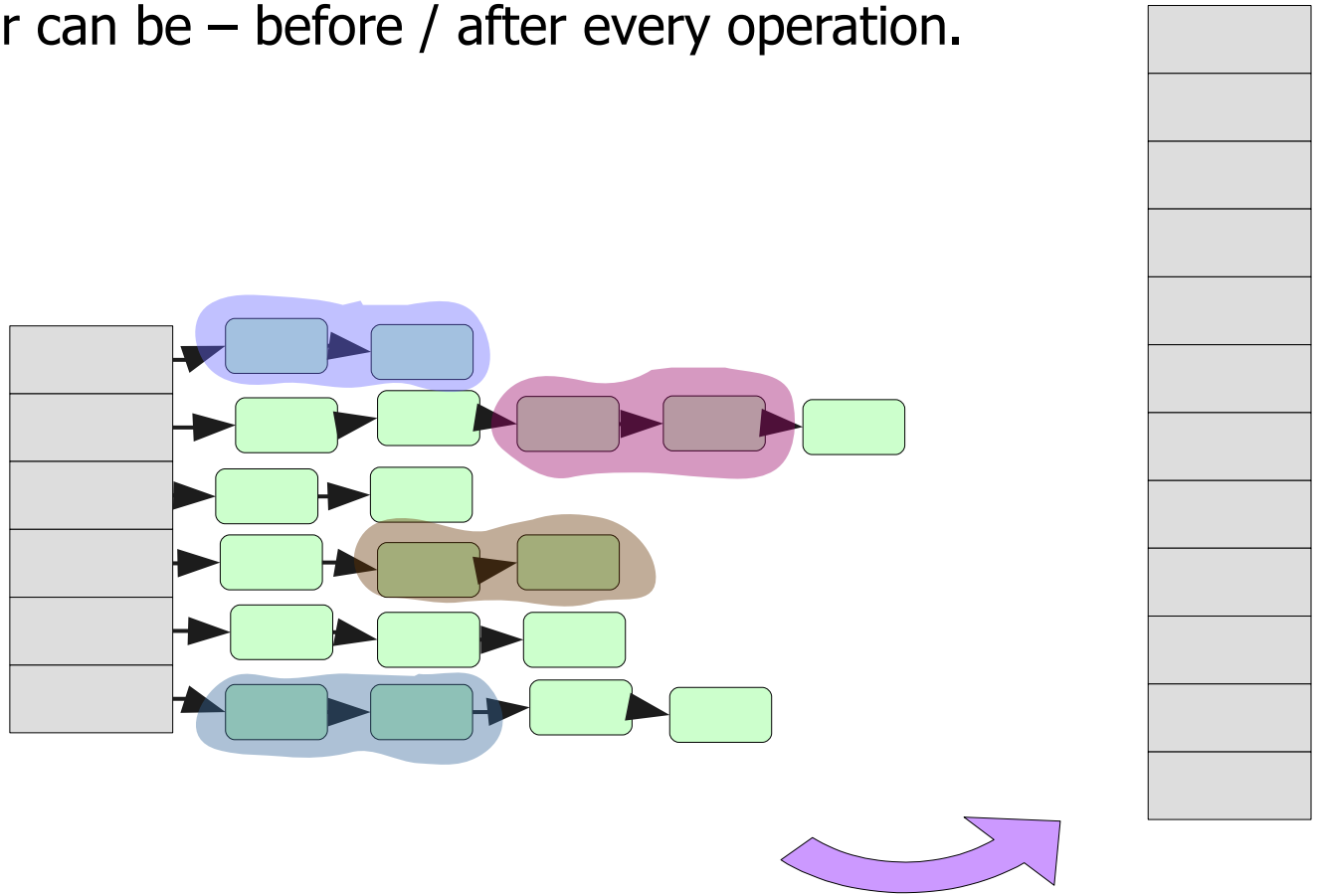


Lazy rehash

That's why practically it is choosen to move **3-5** entries during a step.

The last question: when we should move those portions?

The answer can be – before / after every operation.



Lazy rehash

This way, the final pseudo-code for search becomes:

```
function search( x: Key ) : Value
    if <in intermediate state>
        // Search in old table
        ...
        // If not found, search in new table
        ...
        // Move the next portion
        ...
    else
        // Search in the only table
        ...
```

O(1)

O(1)

Lazy rehash

Final pseudo-code for insertion becomes:

```
procedure insert( x: Key, y: Value )
    if <in intermediate state>
        // Insert right in the new table
        ...
        // Move the next portion
        ...
    else
        // Insert in the only table
        ...
```

O(1)

O(1)

Lazy rehash

Having lazy rehash enabled we can be sure that no operation will ever take **O(N)** time.

... so the server will never act slow, because of this.

Exercises

3) Implement lazy rehash,

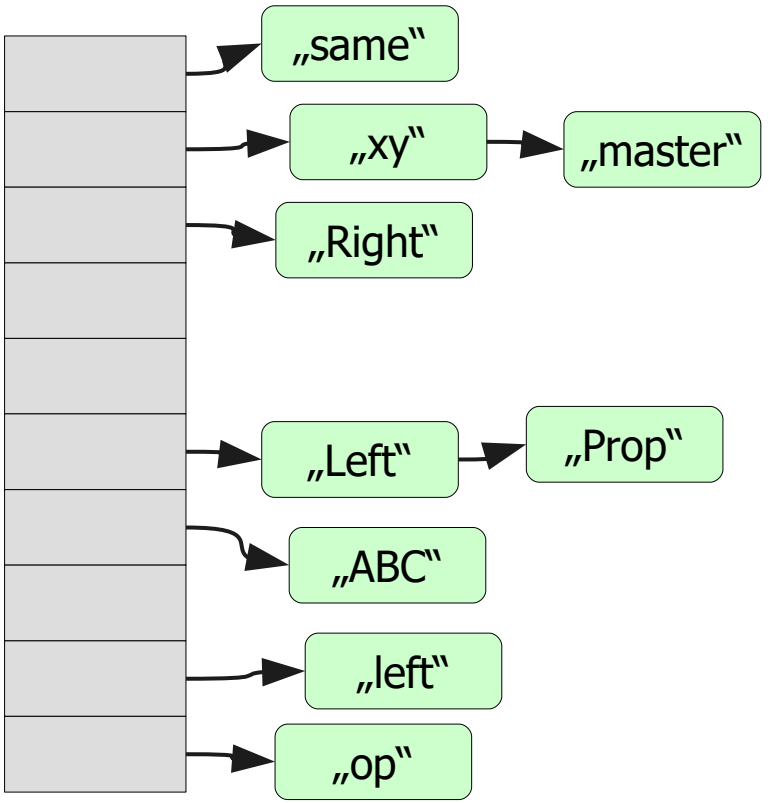
* call it only after insertion,

* call it after every operation on the table.

Open addressing

Closed-addressing is the simplest method for resolving collisions. However, it has some drawbacks:

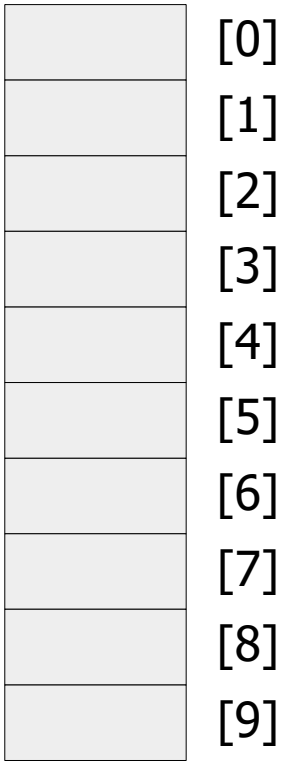
- Traversing linked lists is not cache-optimal, since nodes of the same list might reside in far apart places in the memory.
- Need to allocate / deallocate nodes quite often. Each of those calls addresses the OS.



Open addressing

An alternative, but still popular method for resolving collisions is called "Open adressing".

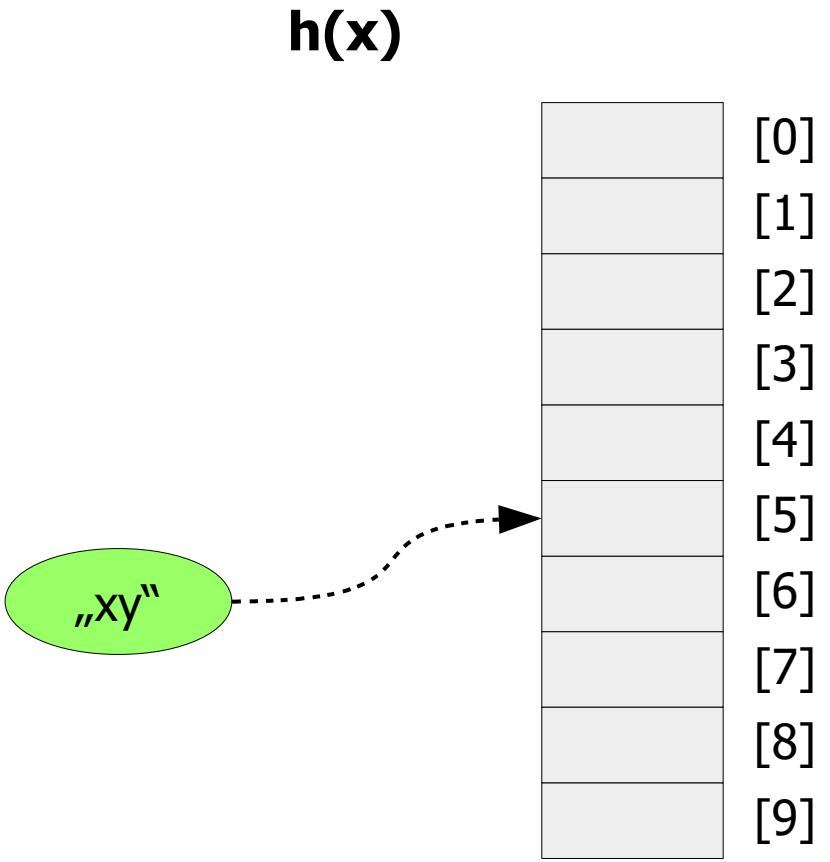
- Here we don't keep array of linked lists, but just an array instead.



Open addressing

An alternative, but still popular method for resolving collisions is called "Open adressing".

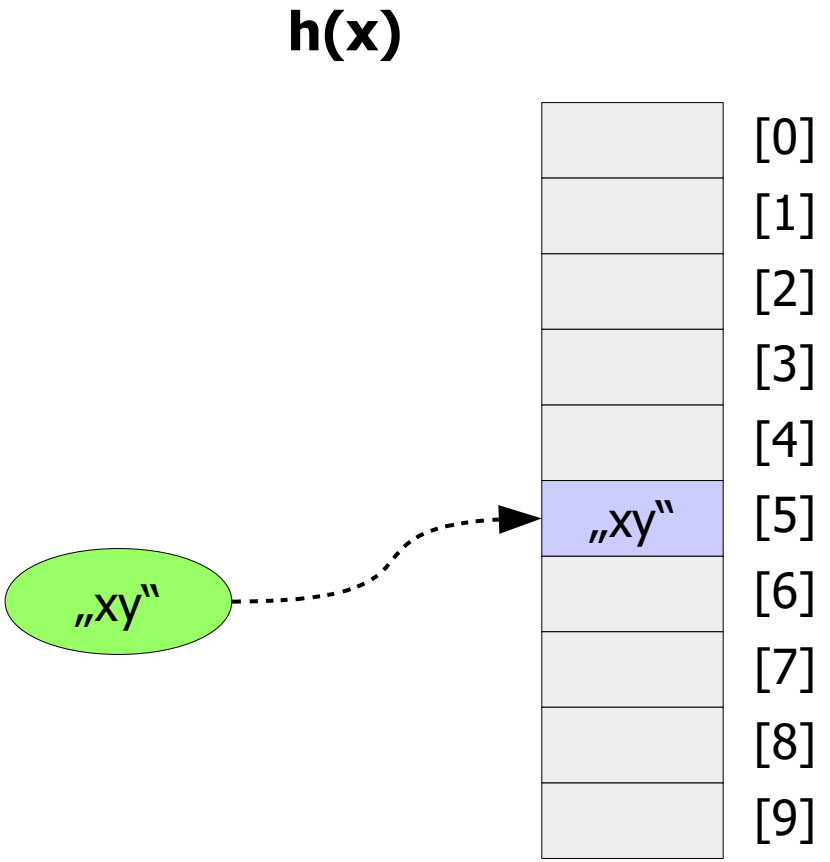
- Here we don't keep array of linked lists, but just an array instead.
- Mapping of values to cells is done in completely the same way, using "**h(x)**".



Open addressing

An alternative, but still popular method for resolving collisions is called "Open adressing".

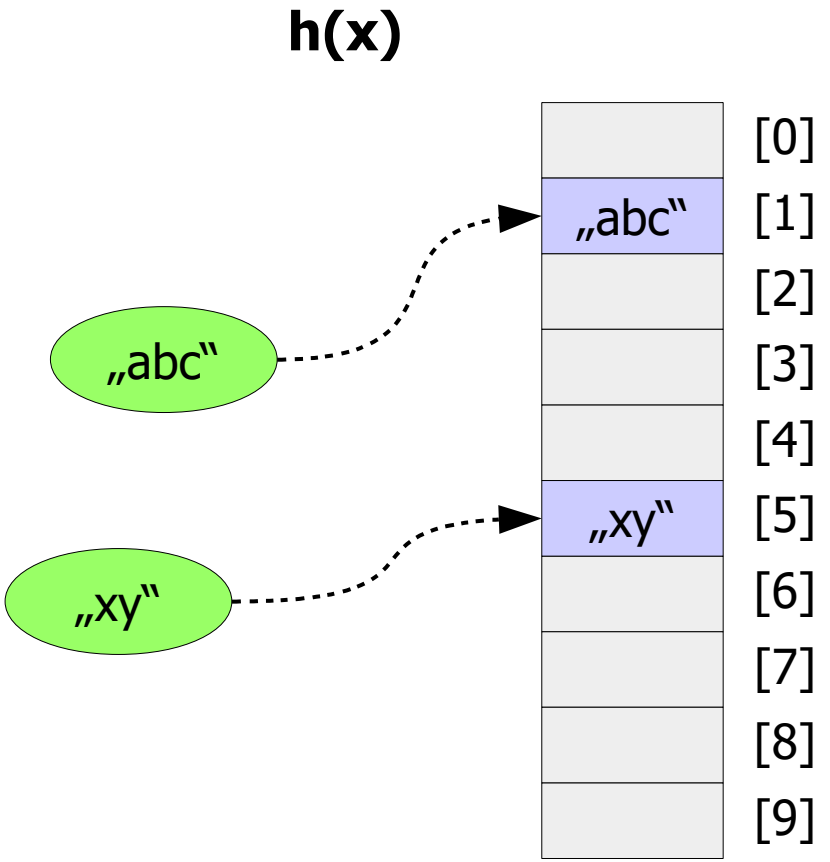
- Here we don't keep array of linked lists, but just an array instead.
- Mapping of values to cells is done in completely the same way, using "**h(x)**".
- During insertion, if the cell is empty, the value is placed there.



Open addressing

An alternative, but still popular method for resolving collisions is called "Open adressing".

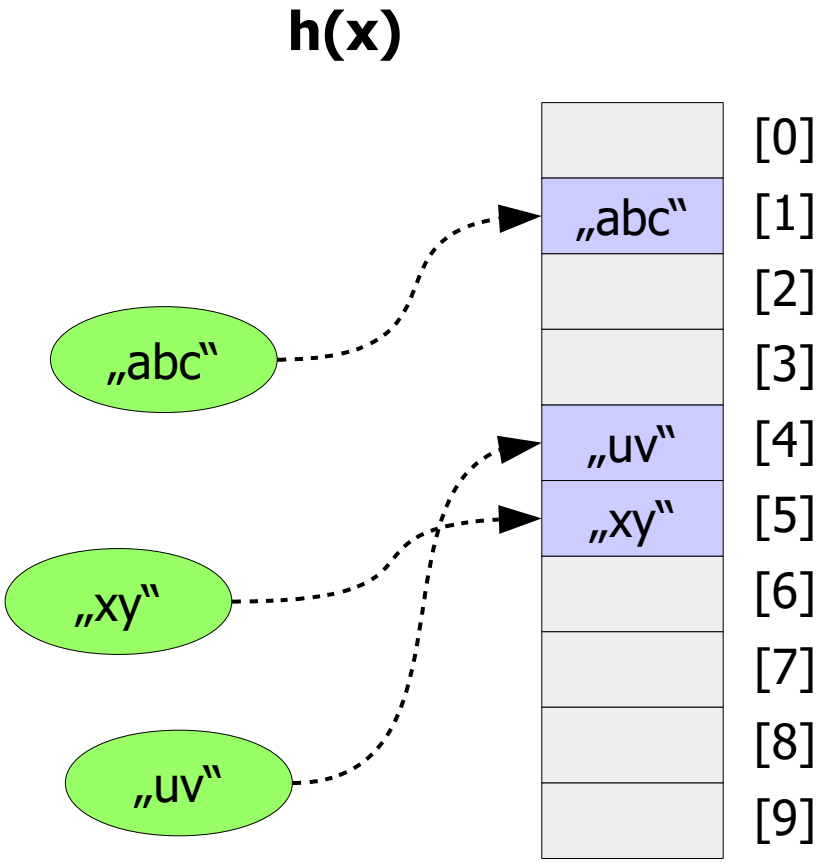
- Here we don't keep array of linked lists, but just an array instead.
- Mapping of values to cells is done in completely the same way, using "**h(x)**".
- During insertion, if the cell is empty, the value is placed there.



Open addressing

An alternative, but still popular method for resolving collisions is called "Open adressing".

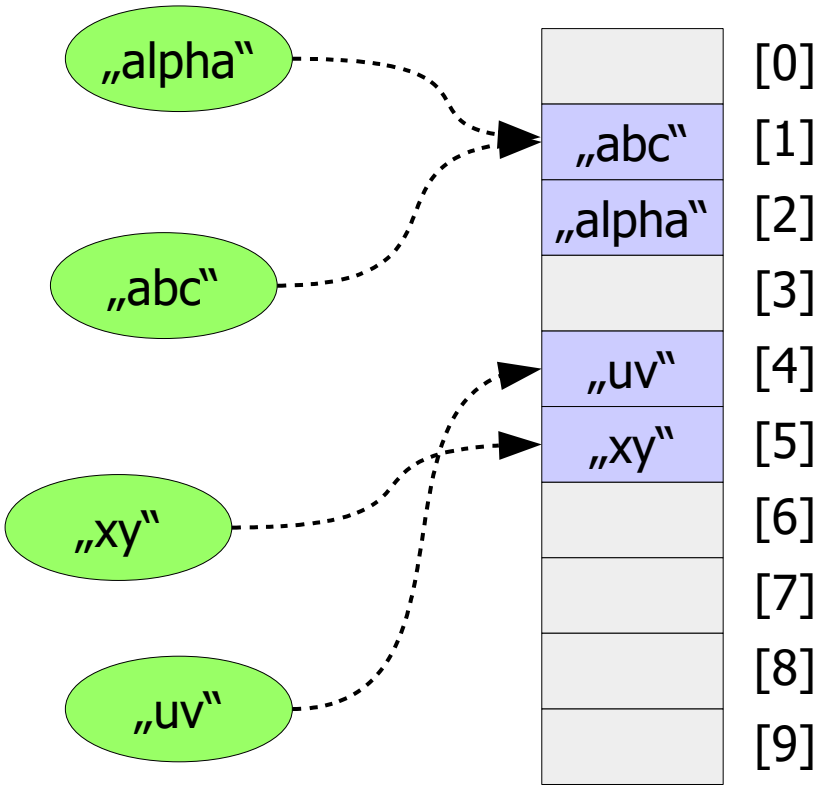
- Here we don't keep array of linked lists, but just an array instead.
- Mapping of values to cells is done in completely the same way, using "**h(x)**".
- During insertion, if the cell is empty, the value is placed there.



Open addressing

An alternative, but still popular method for resolving collisions is called "Open adressing".

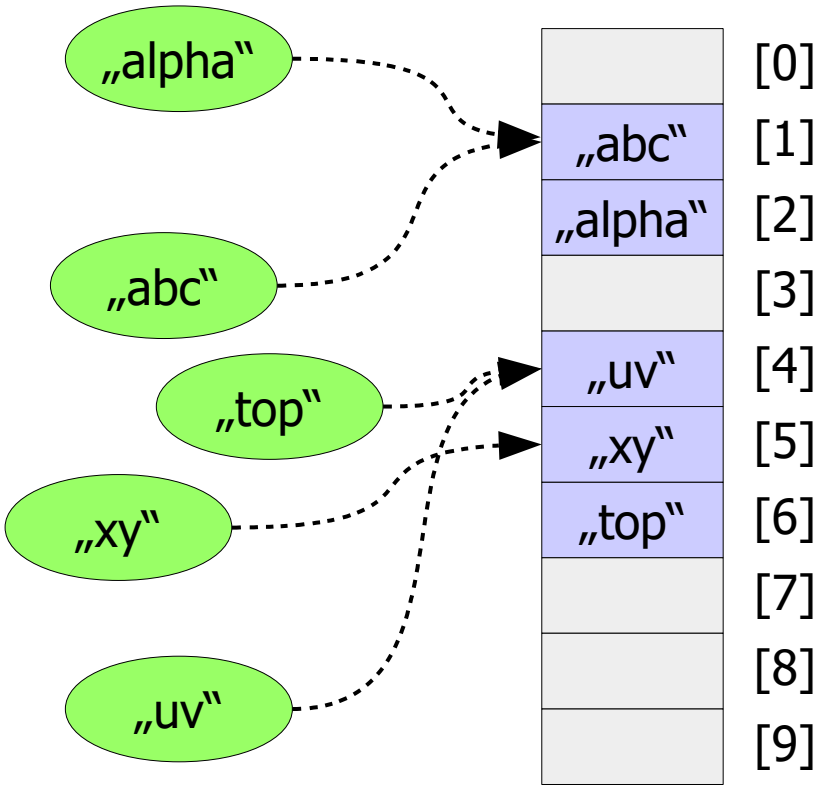
- Here we don't keep array of linked lists, but just an array instead.
- Mapping of values to cells is done in completely the same way, using "**h(x)**".
- During insertion, if the cell is empty, the value is placed there.
- Otherwise, it is placed in the cyclically next empty cell.



Open addressing

An alternative, but still popular method for resolving collisions is called "Open adressing".

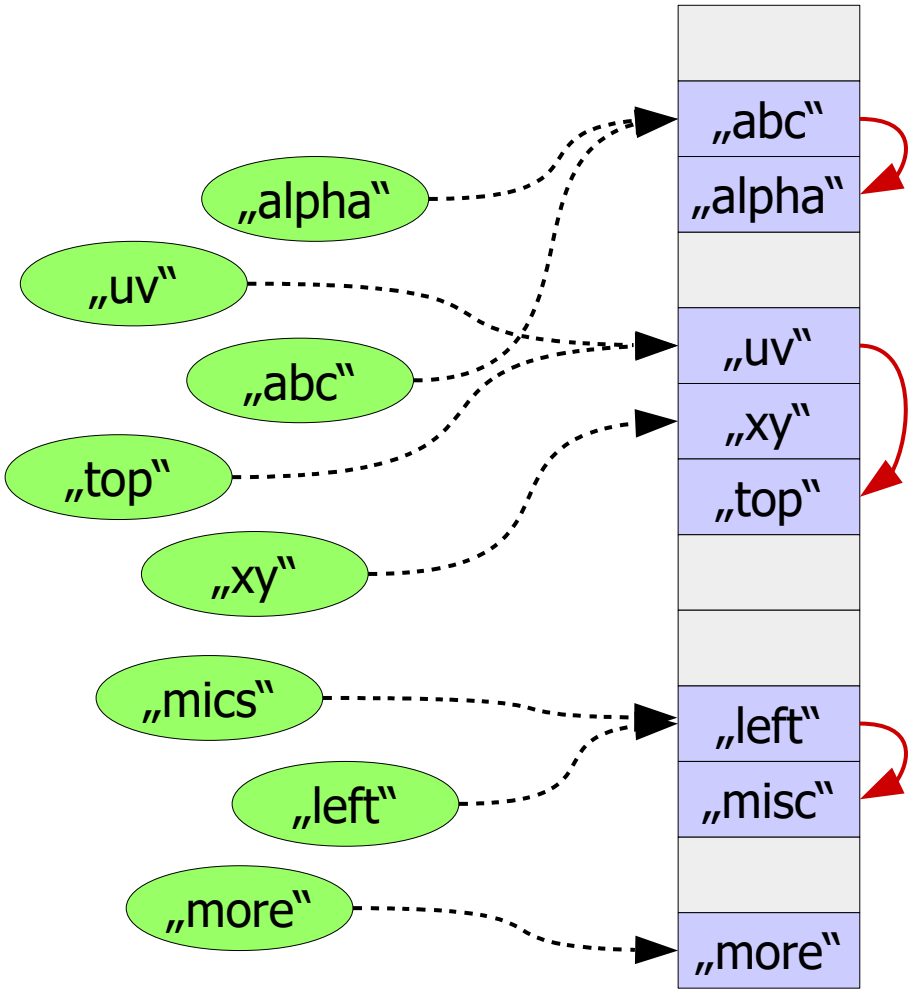
- Here we don't keep array of linked lists, but just an array instead.
- Mapping of values to cells is done in completely the same way, using "**h(x)**".
- During insertion, if the cell is empty, the value is placed there.
- Otherwise, it is placed in the cyclically next empty cell.



Open addressing

How the search should be done?

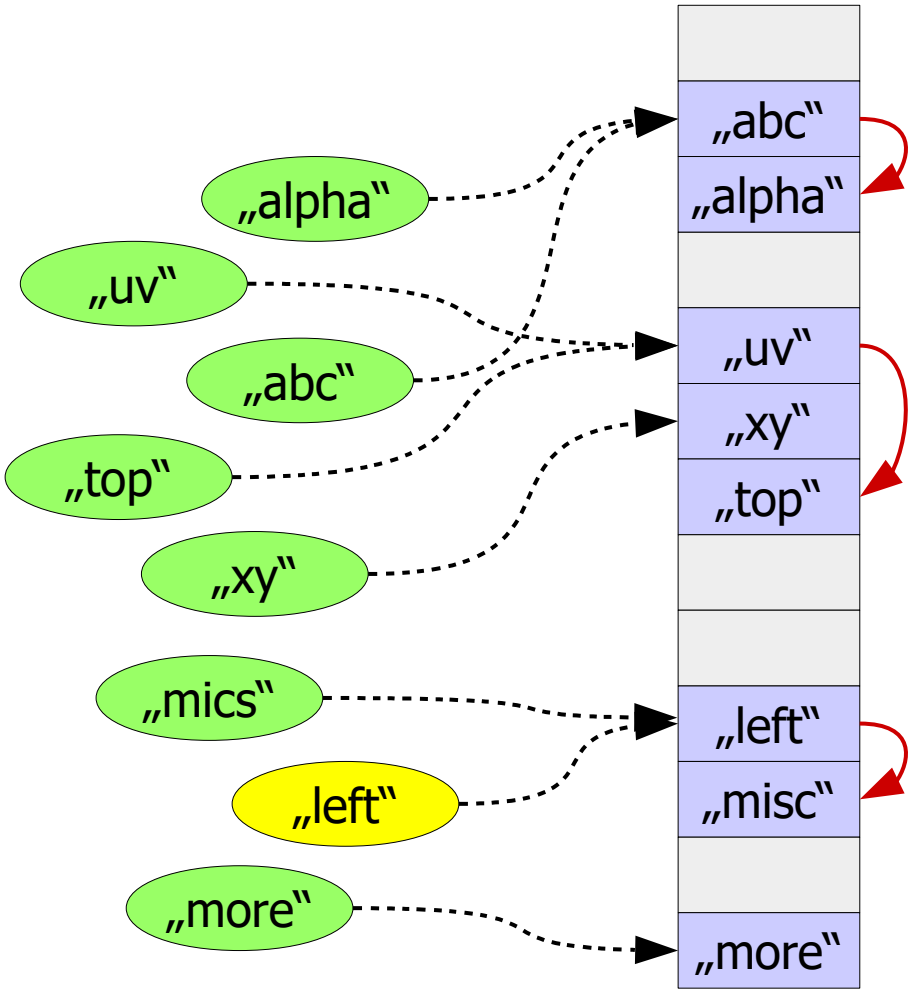
- Any given value "**y**", it might be located either in cell "**h(y)**", or a bit later.



Open addressing

How the search should be done?

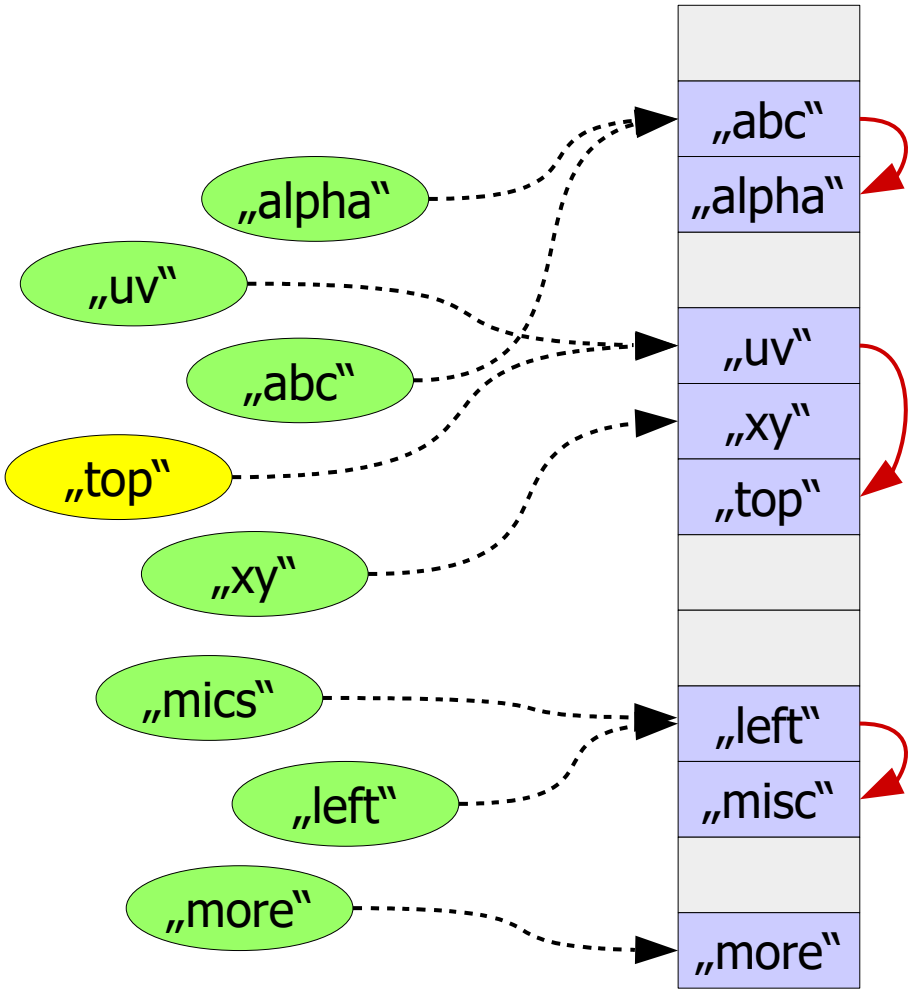
- Any given value "**y**", it might be located either in cell "**h(y)**", or a bit later.
- So similarly to insertion, we locate cell "**h(y)**" and look there.



Open addressing

How the search should be done?

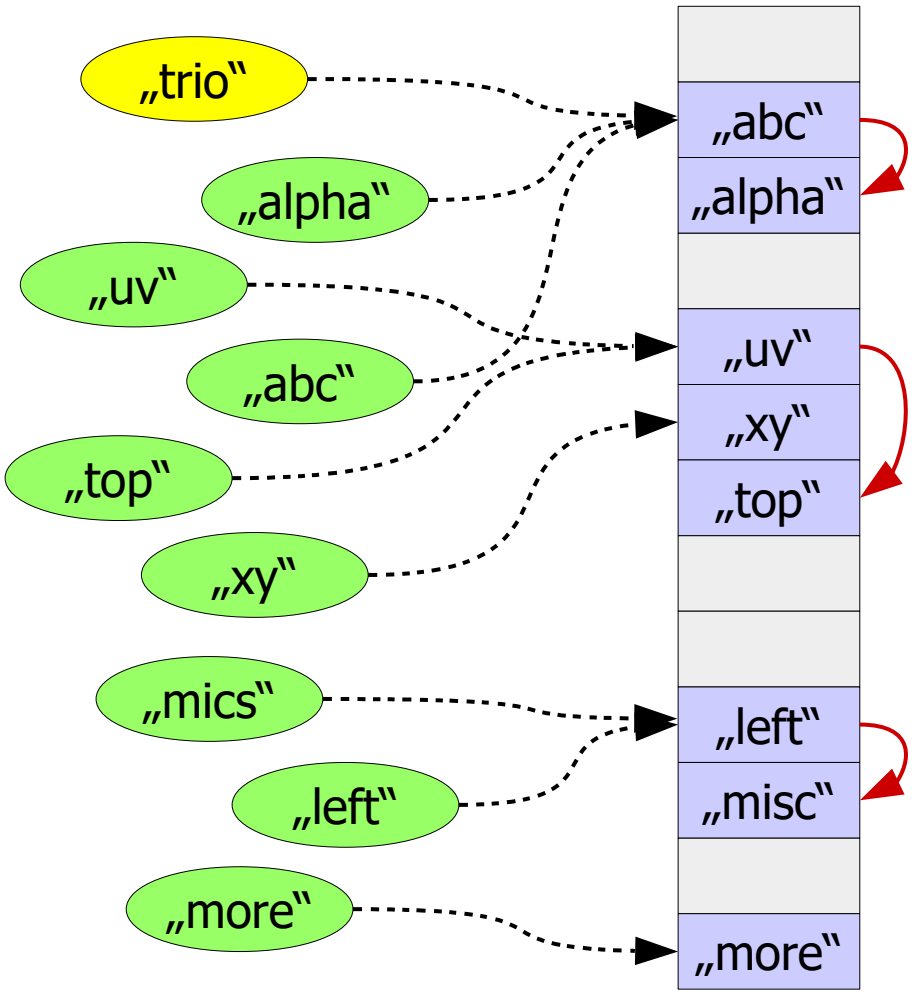
- Any given value "**y**", it might be located either in cell "**h(y)**", or a bit later.
- So similarly to insertion, we locate cell "**h(y)**" and look there.
- If it is not there, we cyclically scan forward.



Open addressing

How the search should be done?

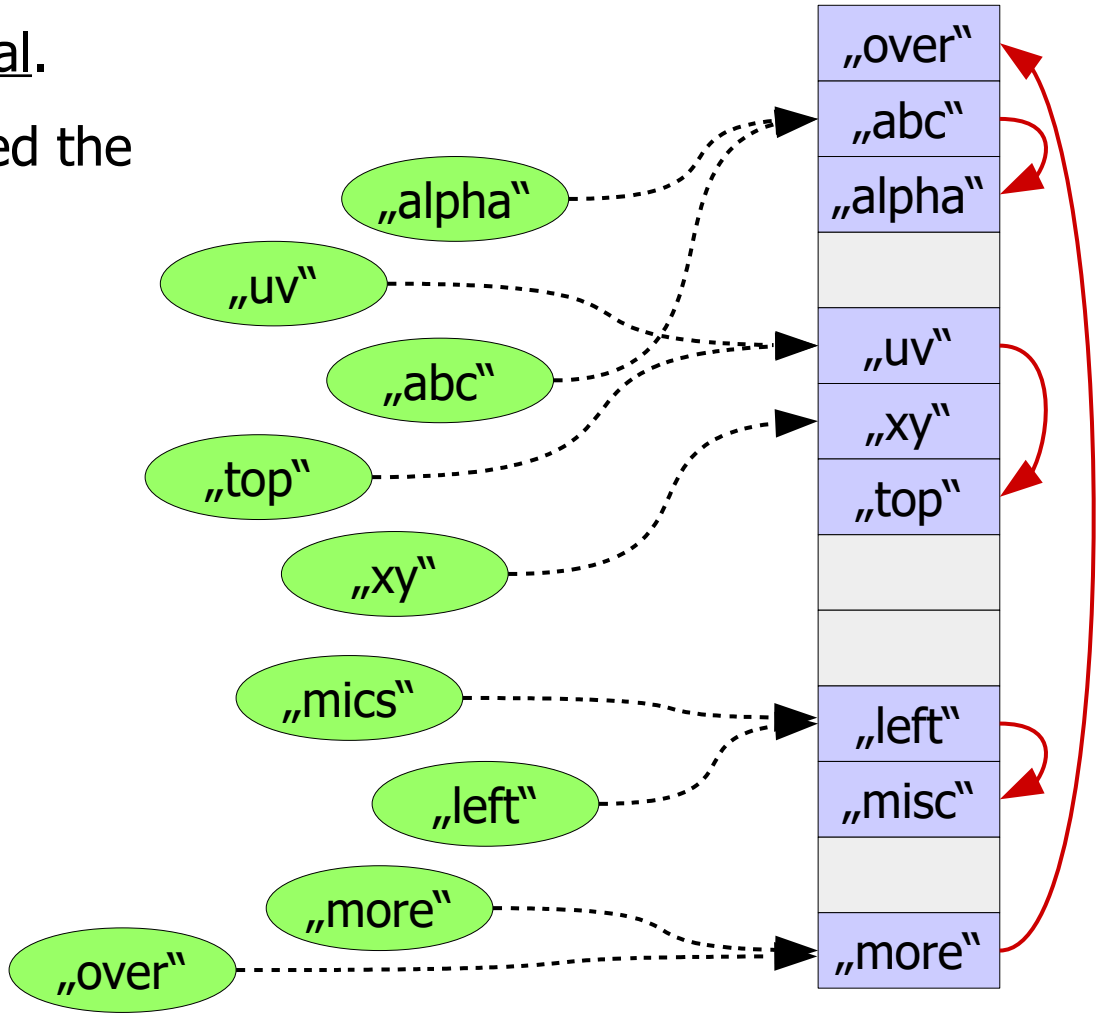
- Any given value "**y**", it might be located either in cell "**h(y)**", or a bit later.
- So similarly to insertion, we locate cell "**h(y)**" and look there.
- If it is not there, we cyclically scan forward.
- Finally, if during scan we reached an empty cell, then '**y**' is absent.



Open addressing

Let's not forget that open-addressing hash table is cyclical.

... so if any scan has reached the end, we continue from the beginning.



Exercise

Implement the following operations for open-addressing hash table:

- Insertion,
- Search.

Open addressing

Question: How do you think, is removal also a simple operation here?

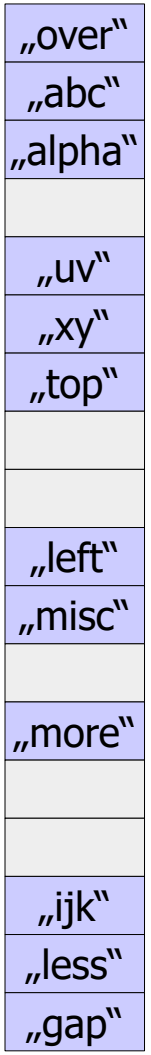
Open addressing

<removal algorithm>

Open addressing

So we have seen that in open-addressing hash table, all **3** algorithms do a linear scan.

... which is why the structure is cache-optimal.



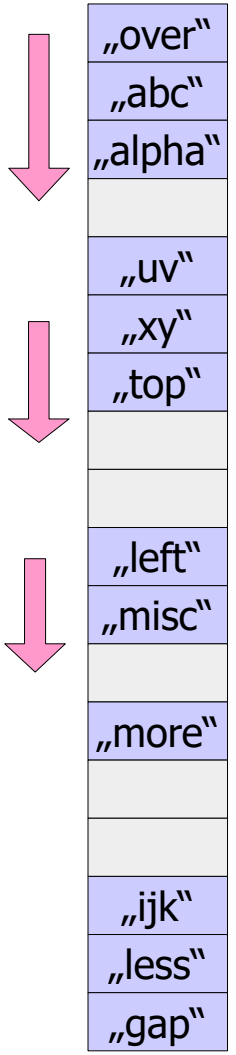
Open addressing

So we have seen that in open-addressing hash table, all **3** algorithms do a linear scan.

... which is why the structure is cache-optimal.

Also, in all the **3** algorithms, there might be need to scan until the next empty cell.

... and we can question, how long it will take to reach an empty cell?



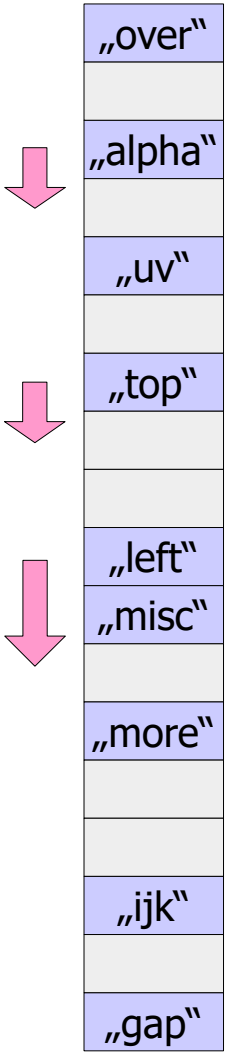
Open addressing

This directly depends on the load factor, which is:

f = N / M.

If **f = 0.5**, almost every second cell will be empty.

... so the scans will be very short.



f = 0.5

Open addressing

This directly depends on the load factor, which is:

f = N / M.

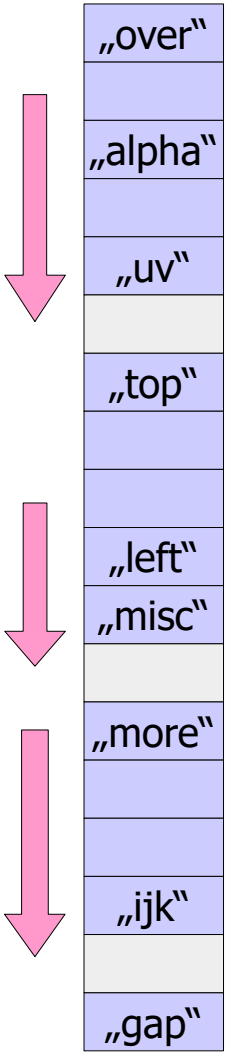
If **f = 0.5**, almost every second cell will be empty.

... so the scans will be very short.

If **f = 0.8**, almost every **5**'th cell will be empty,

... so the scans will be short.

f = 0.8

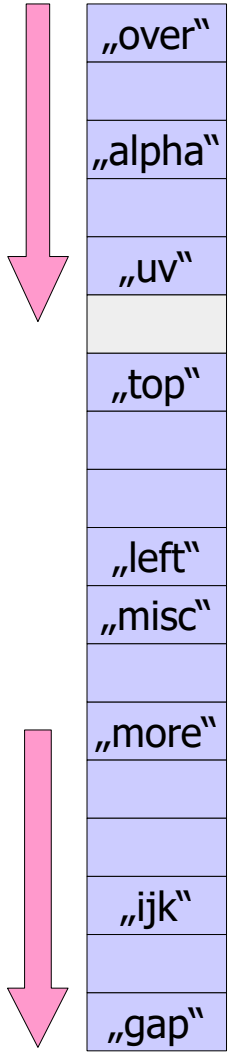


Open addressing

But once **f** reaches **1.0**, lengths of scans increase drastically.

... for example if **f = 0.95**, only one in **20** cells will be empty,

... and the scans become quite long.



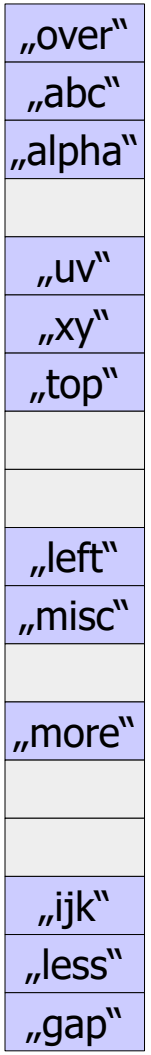
f = 0.95

Open addressing

This is why in open-addressing hash table, a common threshold is about:

f = 0.85

Once it is met, we do rehash.



Exercise

Implement the following operations for open-addressing hash table:

- Rehash,
- Lazy rehash.

Open addressing

<add about primary clustering>

Presentation writer: Tigran Hayrapetyan

Lecturer | Programmer | Researcher

www.linkedin.com/in/tigran-hayrapetyan-cs/

Thank you!

Hash table