Carleton University

Alina Shaikhet

COMP 2402

Hash Tables

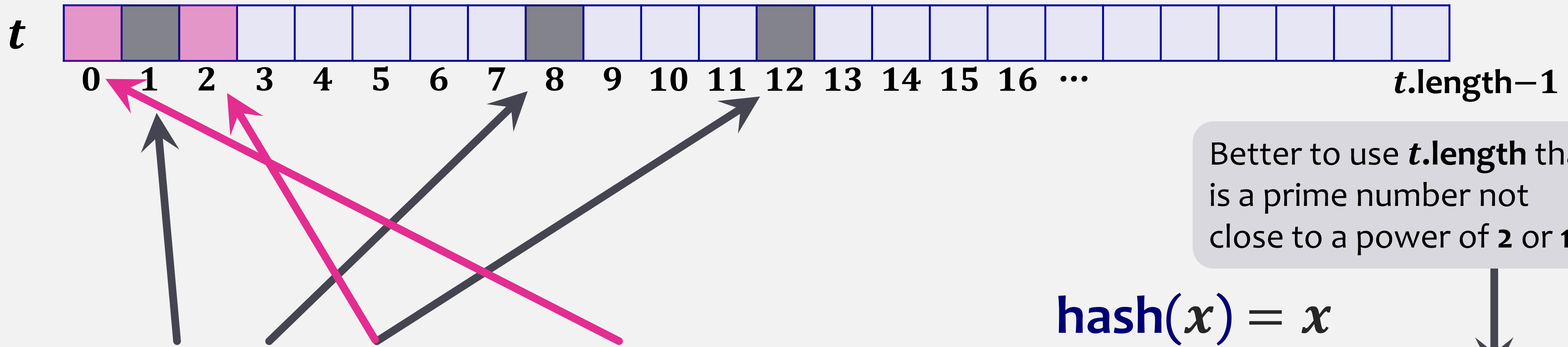
part 1

Intro

$$x = \mathbf{1001110101}$$
$$x \bmod 2^7 = \mathbf{1110101}$$

Hash Tables are data structures with **very fast** insertion and retrieval data.

almost in constant time



$data$: $\mathbf{1, 8, 12, 1000000, 18}$

Better to use $\boldsymbol{t}$**.length** that is a prime number not close to a power of **2** or **10**

$\text{hash}(x) = x$

$\text{hash}(x) = x \bmod 10$

Intro

with every object there is an associated integer (like id)

Intro – Open addressing

- **Linear Probing**

  $\mathbf{hash}(x, l) = (\mathbf{hash}(x, 0) + l\ ) \bmod \boldsymbol{t}\textbf{.length}$

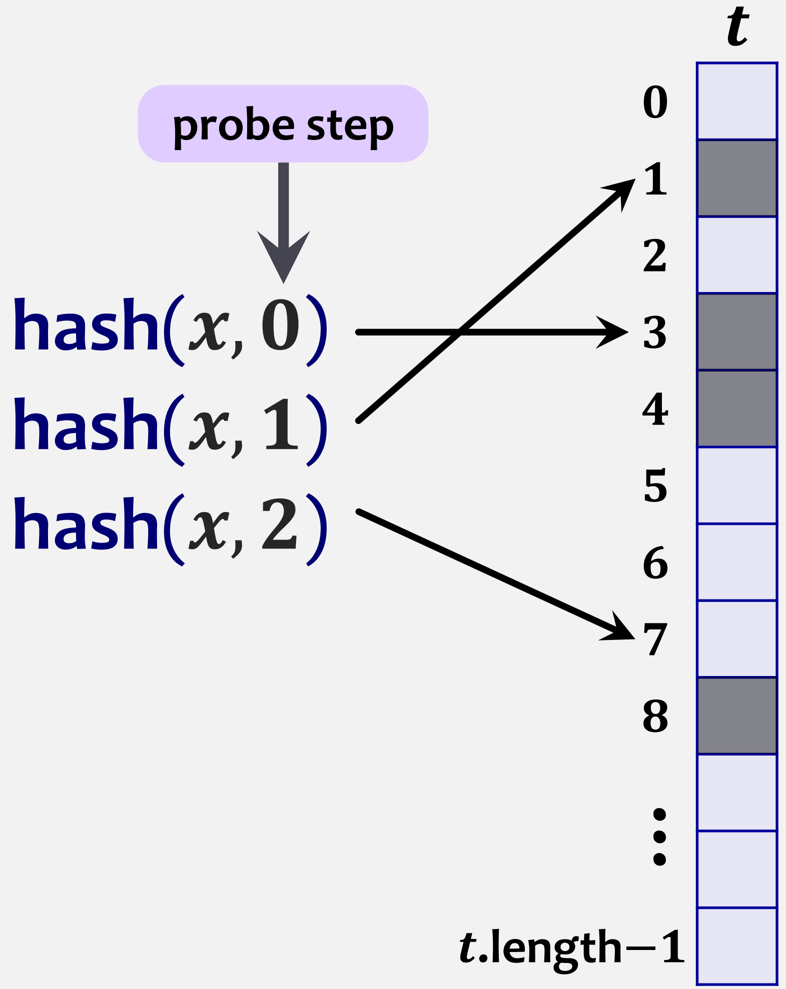
  Problem: **clustering of items**

- **Quadratic Probing**

  $\mathbf{hash}(x, l) = (\mathbf{hash}(x, 0) + l^2\ ) \bmod \boldsymbol{t}\textbf{.length}$

- **Double Hashing**

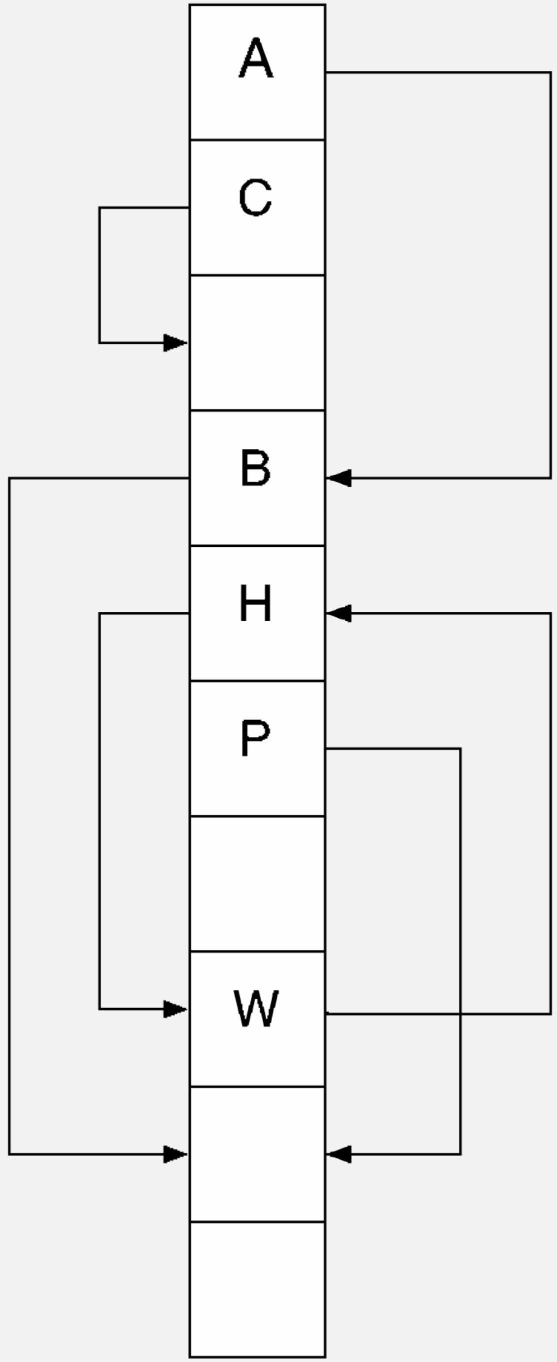
  $\mathbf{hash}(x, l) = (\mathbf{hash}_1(x) + l\ \cdot \mathbf{hash}_2(x)) \bmod \boldsymbol{t}\textbf{.length}$



Problem: deleting an item is not very straightforward

Intro – Open addressing

- **Cuckoo hashing**



https://en.wikipedia.org/wiki/Cuckoo_hashing

Hash Tables

- One of the most used data structuring techniques.
- Largely misunderstood (the most problematic area is computer security)
- Implement **Set** (and **Map**) interface:
  - add(x)
  - remove(x)
  - contains(x)

$O(1)$ expected time

independent of the size of the set

- Make some internal **random** choices.

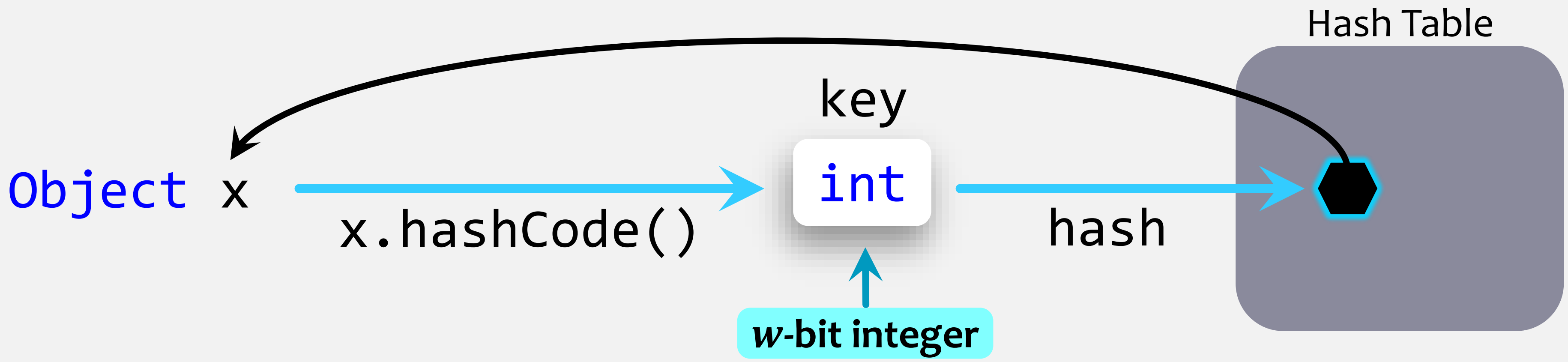
Hash Tables

- Hash tables are an efficient method of storing a small number of **integers** from a large range.
- Very often hash tables store types of data that are **not integers**:

1. Given an object, you associate it with an integer that is suitable for storing in a hash table. In Java: `x.hashCode()`

2. Hash Table will store that integer (together with a reference to x)

hashCode() method (class Object) returns a hash code of the object x



$$\{0, 1, 2, 3, \dots, 2^w - 1\}$$

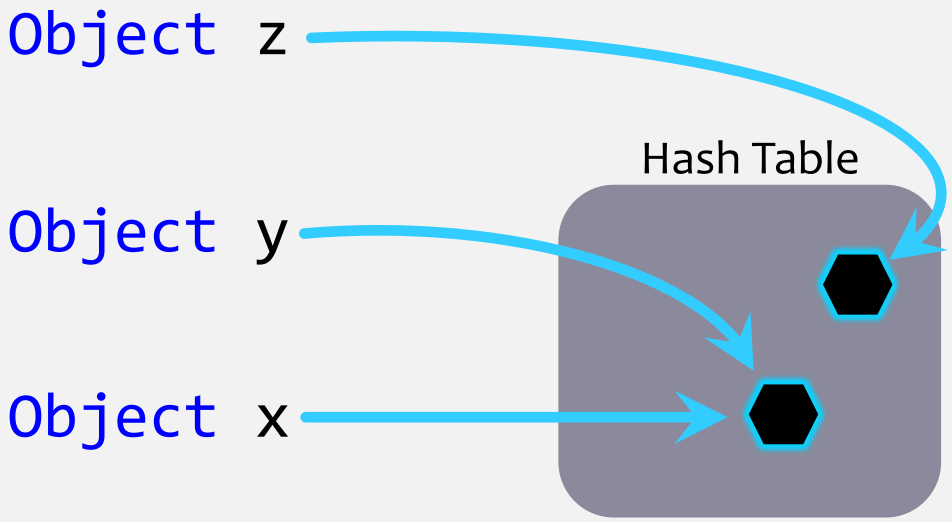
Common Mistake

If two or more objects are equal according to the **`equals`** method (Object class), then their hashes should be equal too:

```
if x.equals(y) then x.hashCode() == y.hashCode()
```

If you override the **equals** method, it is crucial to override the **hashCode** method as well.

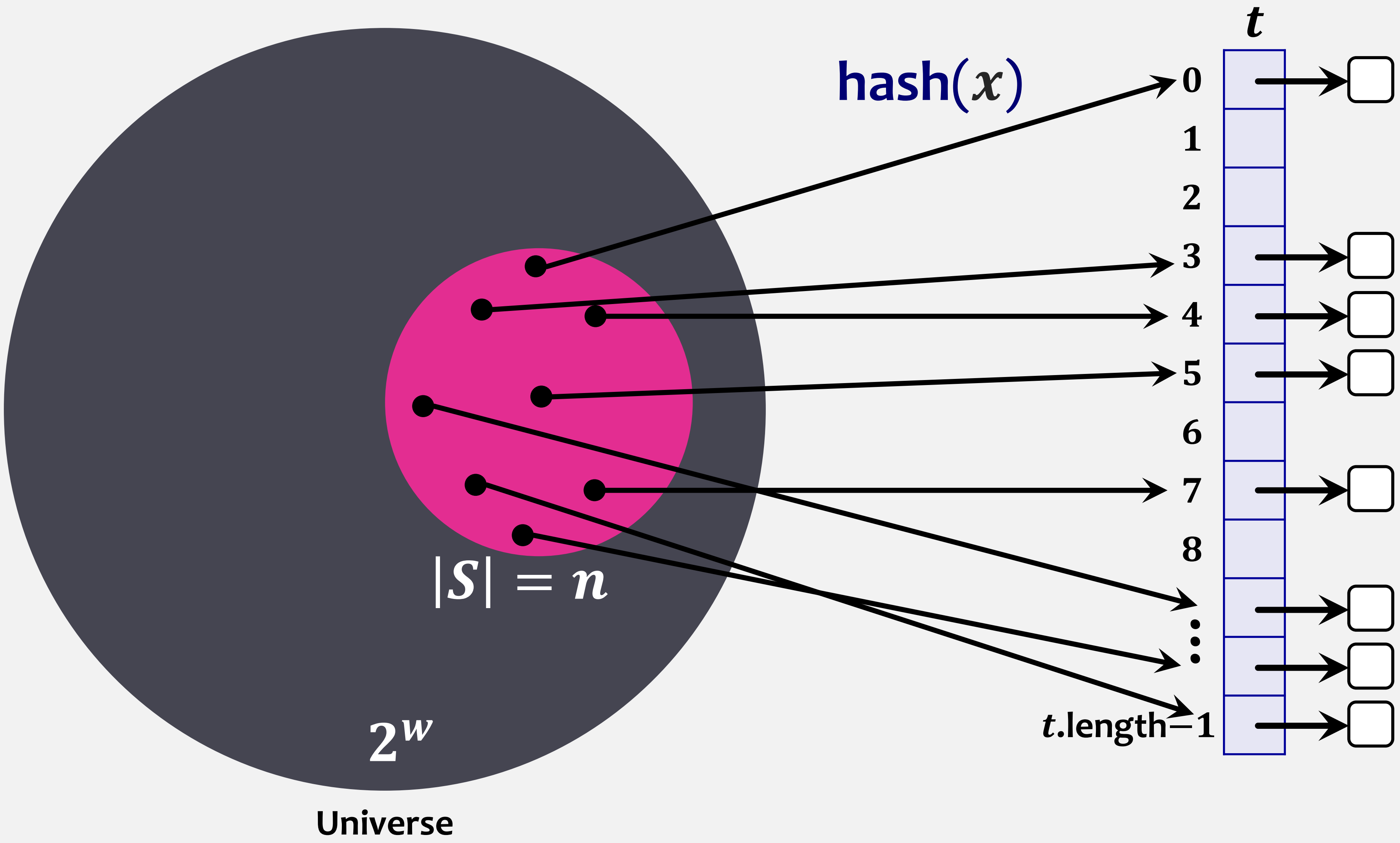
Note: If two objects are **not equal**, we want their hashes to be **not equal**.
But in reality, they can be **equal or unequal**.



Passwords.java & Hashes.java

https://www.educative.io/edpresso/what-is-the-hashcode-method-in-java

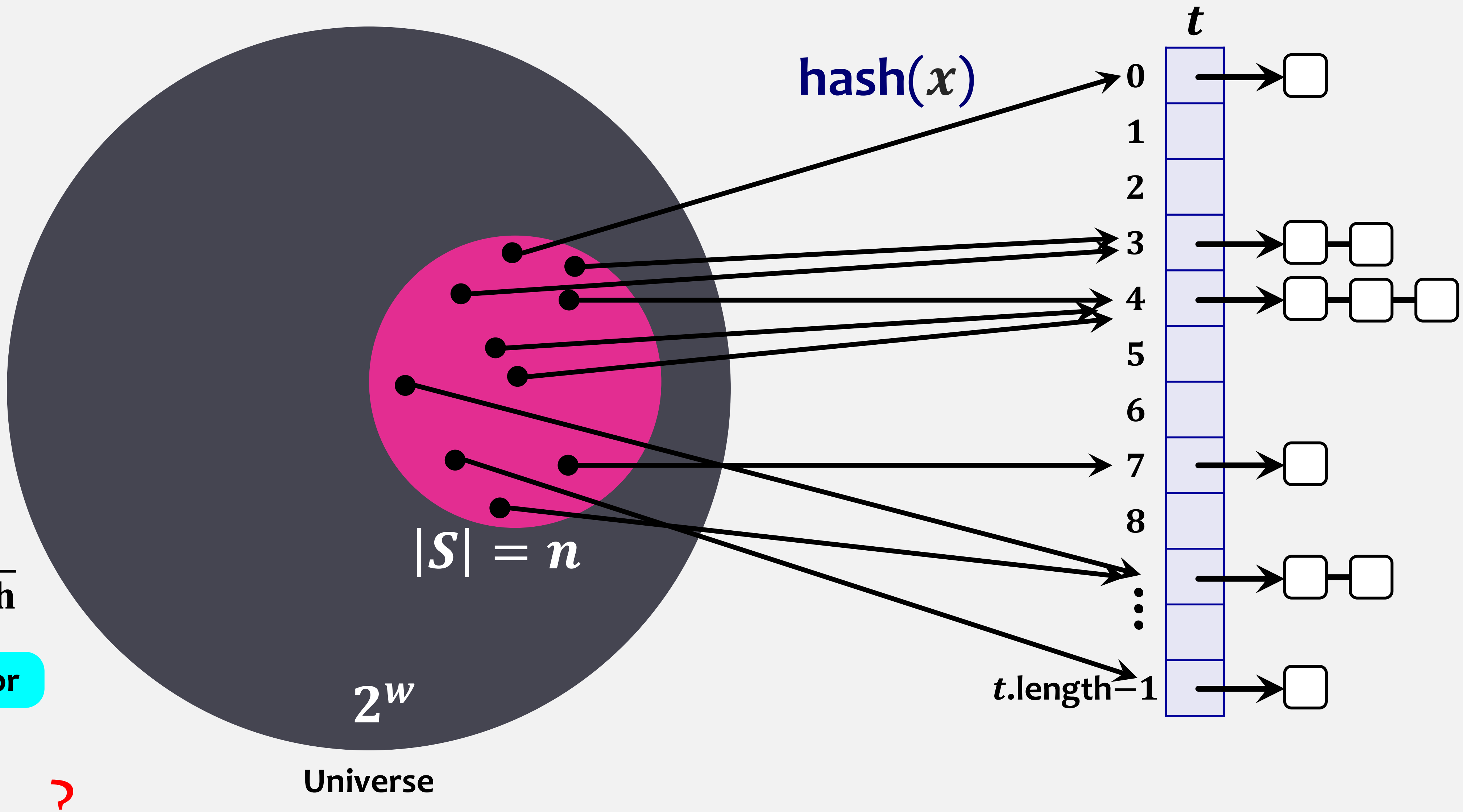
Chained Hash Table



$O(1)$

Ideal situation

Chained Hash Table



for even distribution:

$$\alpha = \frac{n}{t.\,\text{length}}$$

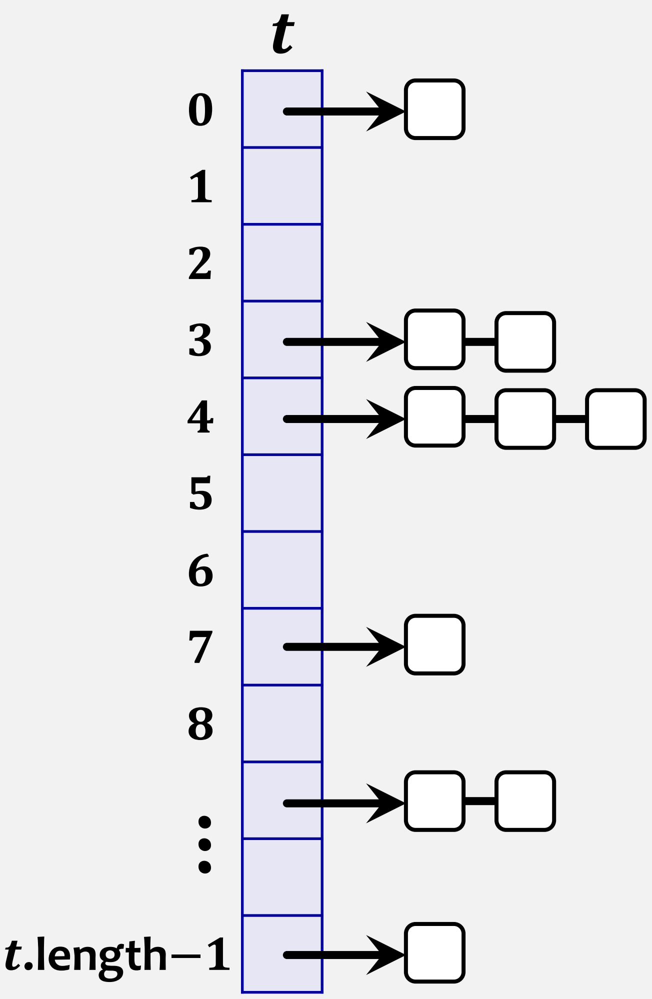
load factor

$$O(1+\alpha) \stackrel{?}{=} O(1)$$

α corresponds to the length of a linked-list

Chained Hash Table

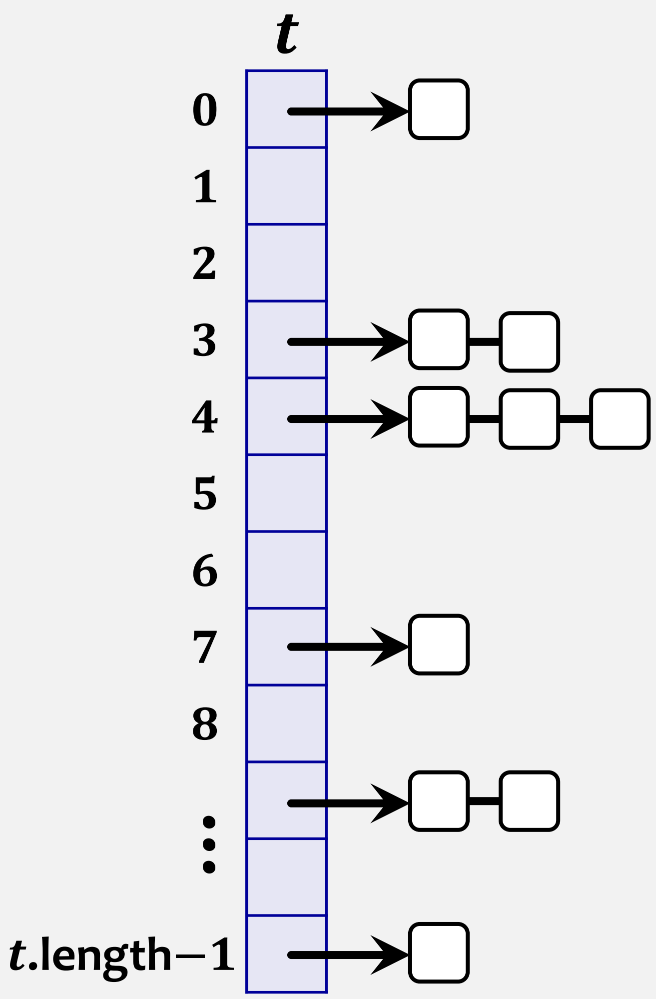
- Assume: the set of objects we're trying to store is a **set of integers** that are all **distinct**.
- Hash table is an array $\boldsymbol{t}$ of lists.
- $\boldsymbol{n}$ is the total number of items in all lists.
- All items with hash value $\boldsymbol{i}$ are stored in the list at $\boldsymbol{t}[\boldsymbol{i}]$.
- function **hash**$(\boldsymbol{x})$ returns the hash value of a data item $\boldsymbol{x}$; $\boldsymbol{x}$ is a $\boldsymbol{w}$-bit integer (for now): $\boldsymbol{x} \in \{\mathbf{0}, \mathbf{1}, \dots, \mathbf{2}^{\boldsymbol{w}} - \mathbf{1}\}$; hash value $\boldsymbol{i}$ is in the range $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \dots, \boldsymbol{t}.\textbf{length} - \mathbf{1}\}$.
- for lists not to get too long, we maintain $\boldsymbol{n} \leq \boldsymbol{t}.\textbf{length}$
- average number of elements stored in one of these lists is $$\frac{\boldsymbol{n}}{\boldsymbol{t}.\textbf{length}} \leq \mathbf{1}$$



contains($\boldsymbol{x}$)

We perform a linear search on the list $\boldsymbol{t}$[hash($\boldsymbol{x}$)]:

```
T contains(x):
    i = hash(x);
    list = t[i];
    for each (y in list)
        if (y.equals(x)) then
            return y;
    return null;
```



$O(n_{\text{hash}(x)})$

Where $n_{\text{hash}(x)}$ is the size of the list t[hash(x)]

add(x)

the cost of growing is only constant when amortized over a sequence of insertions

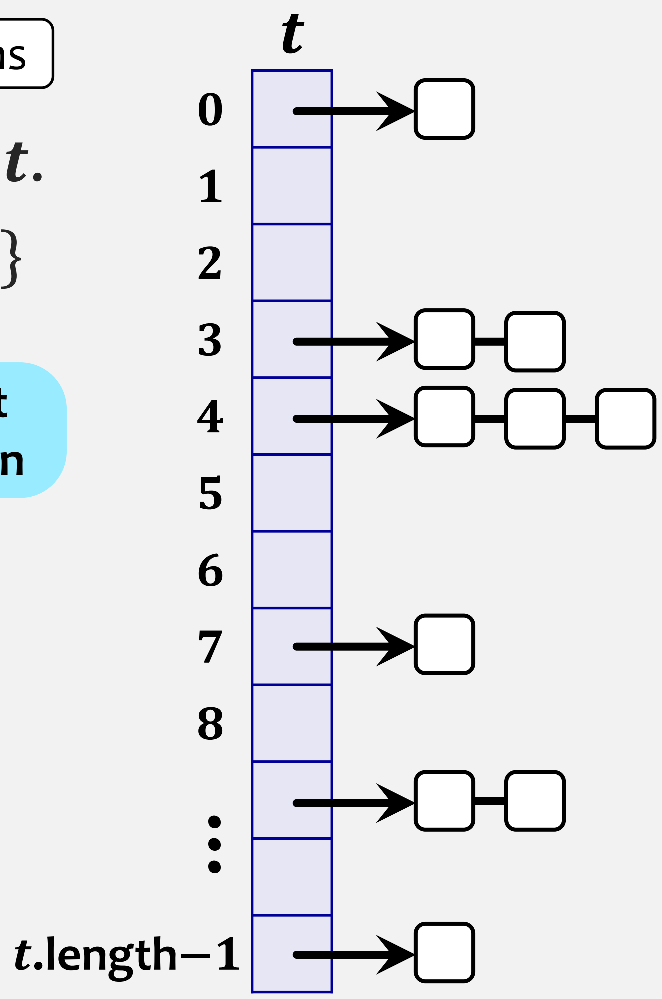
- $O(n)$ • If the length of t needs to be increased, then grow t.
- $O(1)$ • hash x to get an integer $i \in \{0, 1, 2, \dots, t.\text{length} - 1\}$
- $O(1)$ • append x to the list $t[i]$

remove this if you want to store the same x again

```
boolean add(x):
    if (contains(x) ≠ null) then  return false;
    if (n + 1 > t.length) then  resize();
    t[hash(x)].add(x);
    n++;
    return true;
```

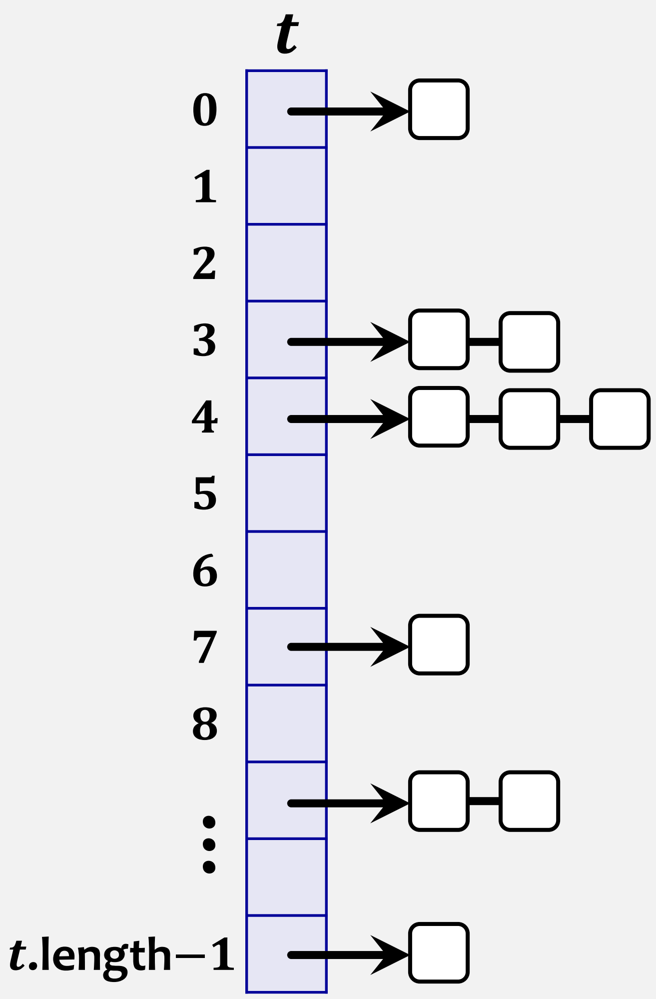
$i = \text{hash}(x)$;
list $= t[i]$;
list.add(x);

$O(n_{\text{hash}(x)})$



remove(x)

- hash x to get an integer $i \in \{0, 1, 2, \ldots, t.\text{length} - 1\}$
- iterate over the list $t[i]$ until you find x, and remove it
- (optional) if the length of t needs to be decreased, then shrink t.



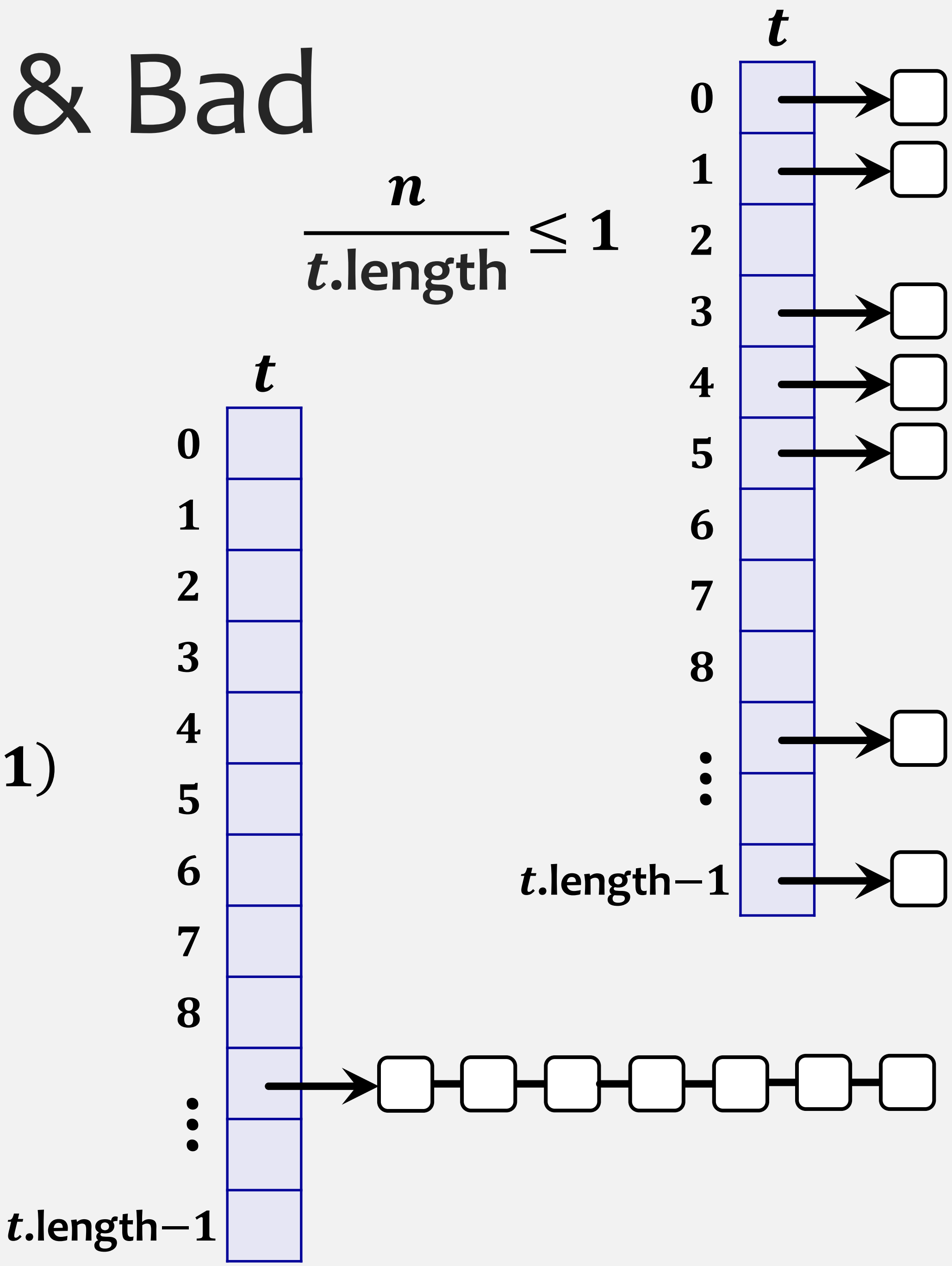
T remove(x):
    if (contains(x) = null) then return null;
expensive $y = t[\text{hash}(x)].\text{remove}(x)$;
    $n--$;
    if ($4n < t.\text{length}$) then resize();
    return y;

$O(n_{\text{hash}(x)})$

Where $n_{\text{hash}(x)}$ is the size of the list $t[\text{hash}(x)]$

Hash Functions – Good & Bad

$$\frac{n}{t.\text{length}} \le 1$$



The performance of a hash table depends on the choice of the hash function.

- A **good** hash function will spread the elements evenly among the $t.\text{length}$ lists, so the expected size of the list $t[\text{hash}(x)]$ is

$$O\left(\frac{n}{t.\text{length}}\right) = O(1)$$

  good hash function should not depend on patterns in the data

- A **bad** hash function will hash all values (including x) to the same table location, so the size of the list $t[\text{hash}(x)]$ will be n.

Universal Hashing

We select a hash function **at random** from a family of hash functions with a certain mathematical property. This guarantees a low number of collisions in expectation.

We want for any $\boldsymbol{x}, \boldsymbol{y} \in \{\mathbf{0}, \dots, \mathbf{2}^{\boldsymbol{w}} - \mathbf{1}\}$, $\boldsymbol{x} \neq \boldsymbol{y}$ $\quad \boldsymbol{Pr}(\text{hash}(\boldsymbol{x}) = \text{hash}(\boldsymbol{y})) \leq \dfrac{\mathbf{1}}{\boldsymbol{t}.\,\textbf{length}}$

Example of a family of hash functions:

Let $\boldsymbol{z}$ be a random number in $\{\mathbf{0}, \dots, \mathbf{2}^{\boldsymbol{w}} - \mathbf{1}\}$, and let $\boldsymbol{t}.\,\textbf{length}$ be a prime number.

Then the formula for hashing an integer $\boldsymbol{x}$ is

Carter & Wegman - 1979

$$\text{hash}(x) = (z \cdot x) \ \textbf{mod} \ \boldsymbol{t}.\,\textbf{length}$$

expensive

“Universal classes of hash functions” by J. Lawrence Carter & Mark N. Wegman

Alina Shaikhet – COMP2402 – Carleton University

Family of hash functions

$$\text{hash}(x) = (z \cdot x) \ \mathbf{mod} \ t.\mathbf{length}$$

Improvement: let t.**length** be 2^k (for some integer k)

$$y \ \mathbf{mod} \ 2^k \quad \equiv \quad \text{last } k \text{ bits of } y$$

Problem: there are certain sets of integers that are hashed with lots of collisions. With this hash function you can get lists of logarithmic length.