Lecture 3

Hashing

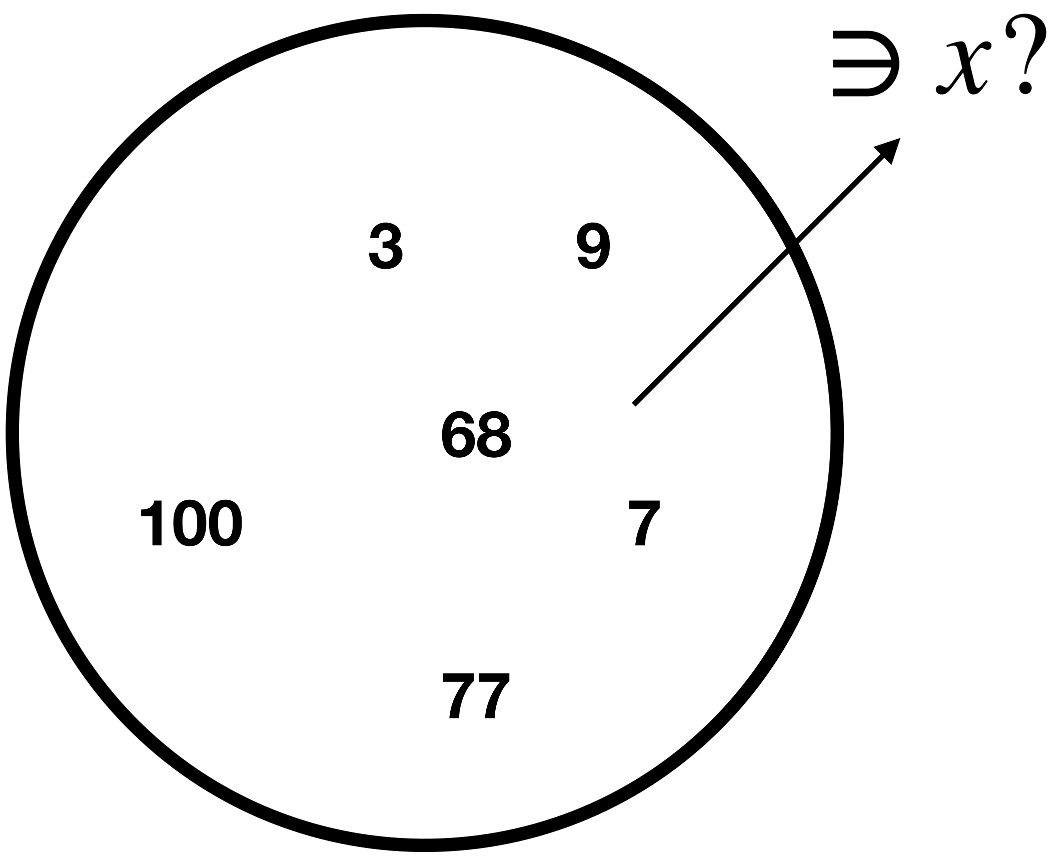
ÉCOLE NORMALE SUPÉRIEURE

PSL RESEARCH UNIVERSITY PARIS

Today's plan

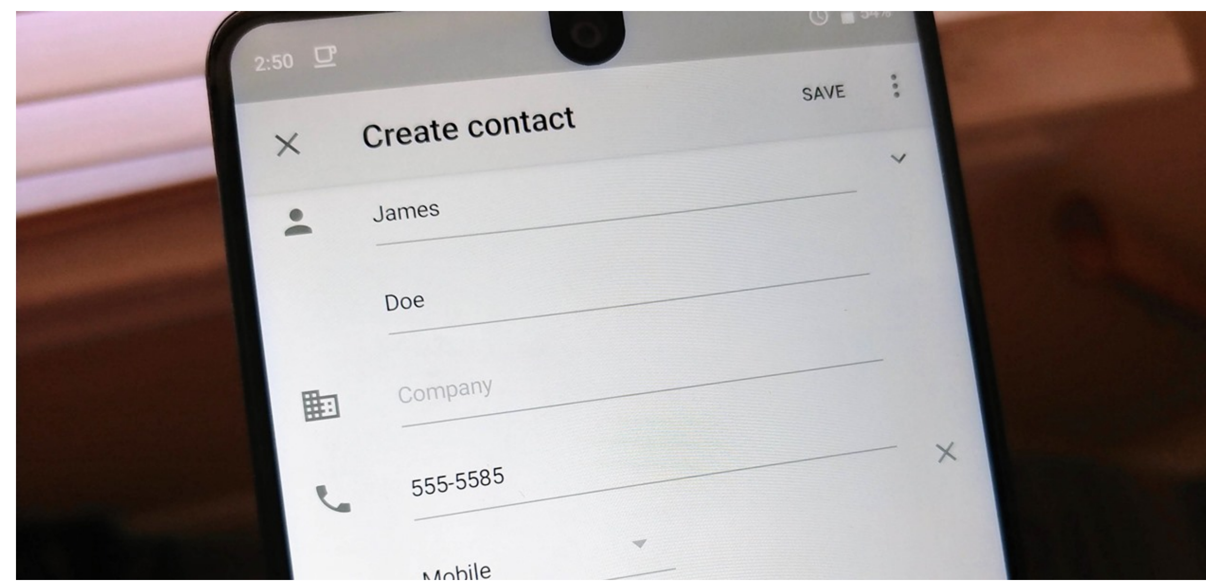
1. Chained hash tables
2. Designing hash functions
3. Open addressing
4. Cuckoo hashing
5. Rolling hash functions

Sets and dictionaries



Set

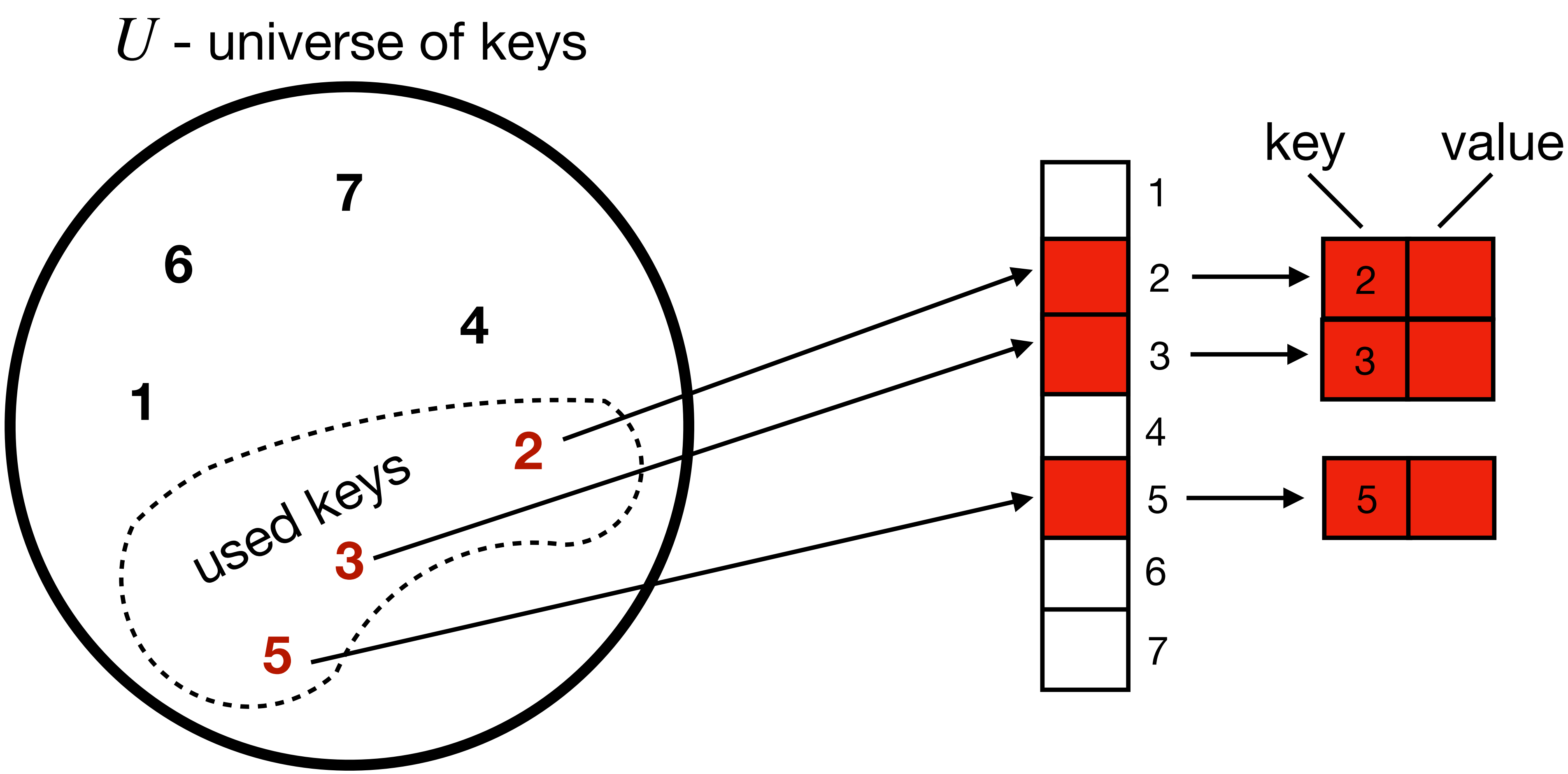
Dictionary



What's the phone number of x?

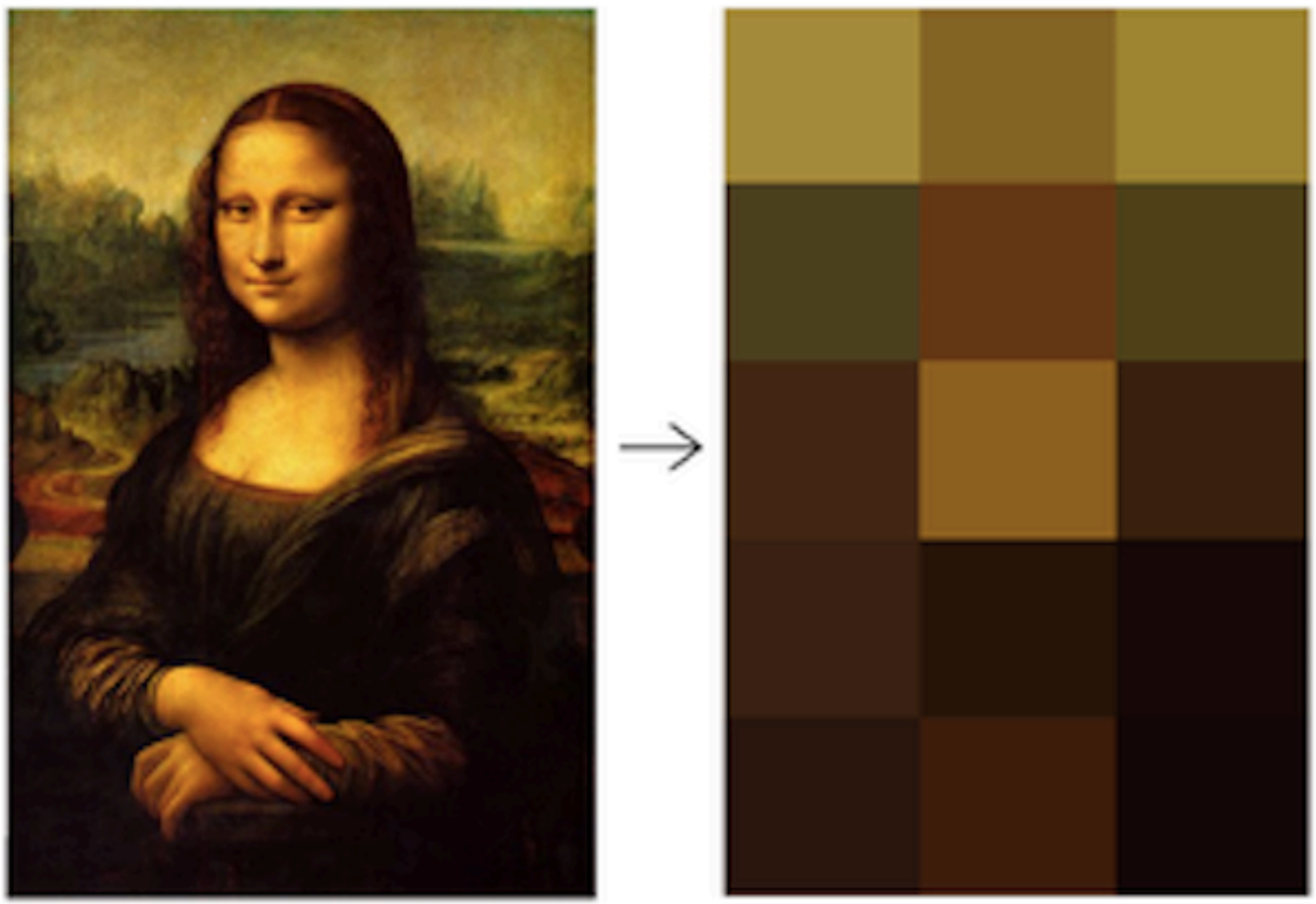
- **Python:** lists, dictionaries
- **C++:** set, map

Array-based implementation



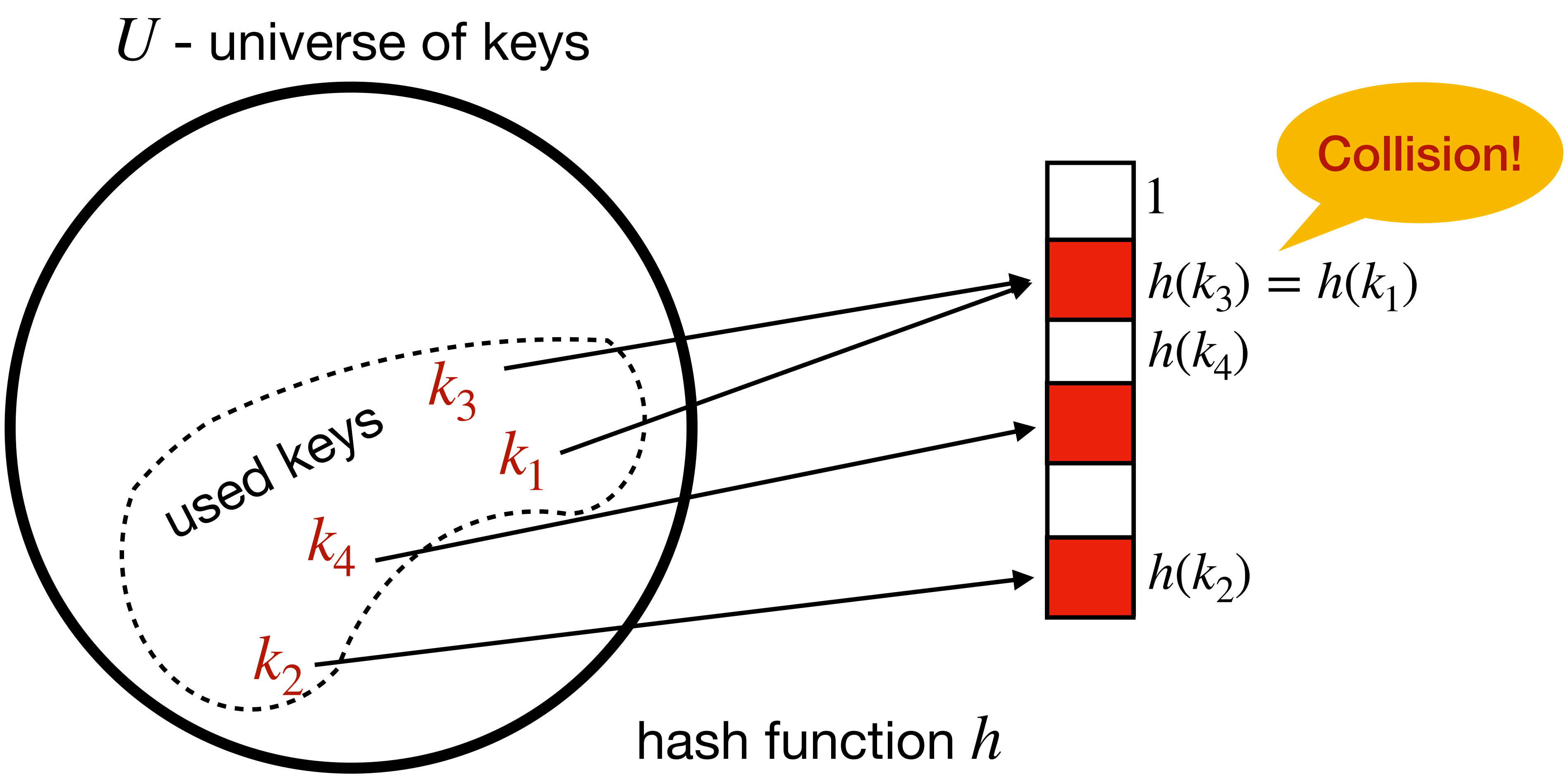
- **Assumption:** no two objects have equal keys
- Space $O(|U|)$, search time $O(1)$, insert / delete time $O(1)$

Chained hash tables

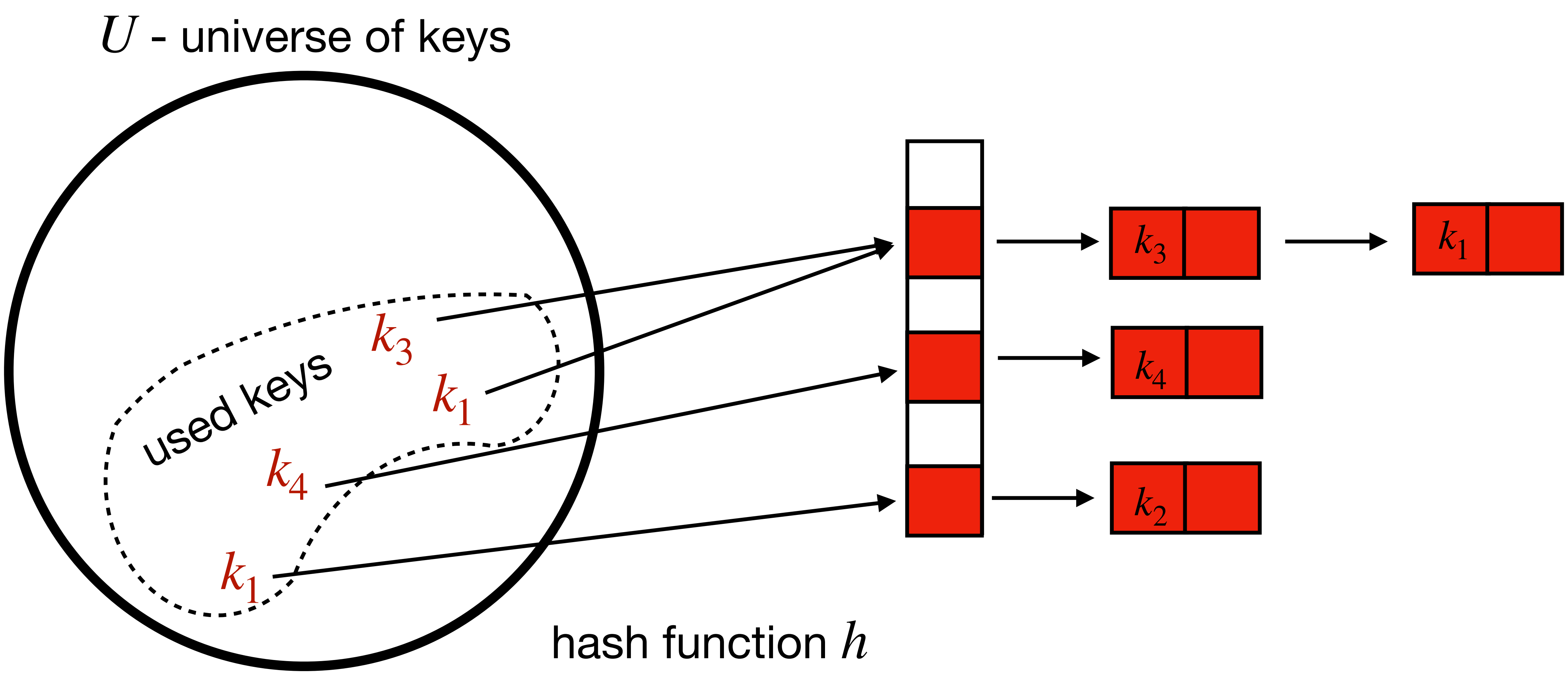


hash function representing an image by a low-res image

Chained hash tables

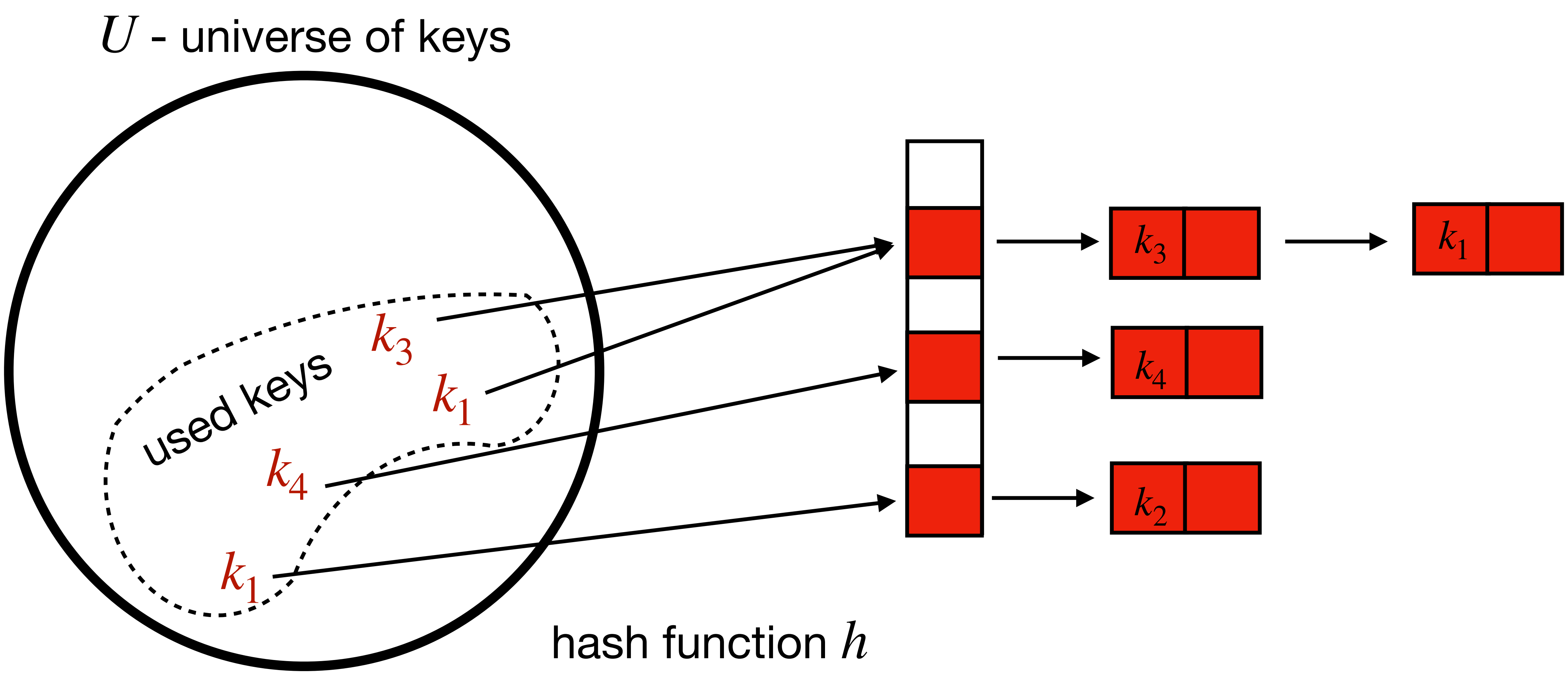


Chained hash tables



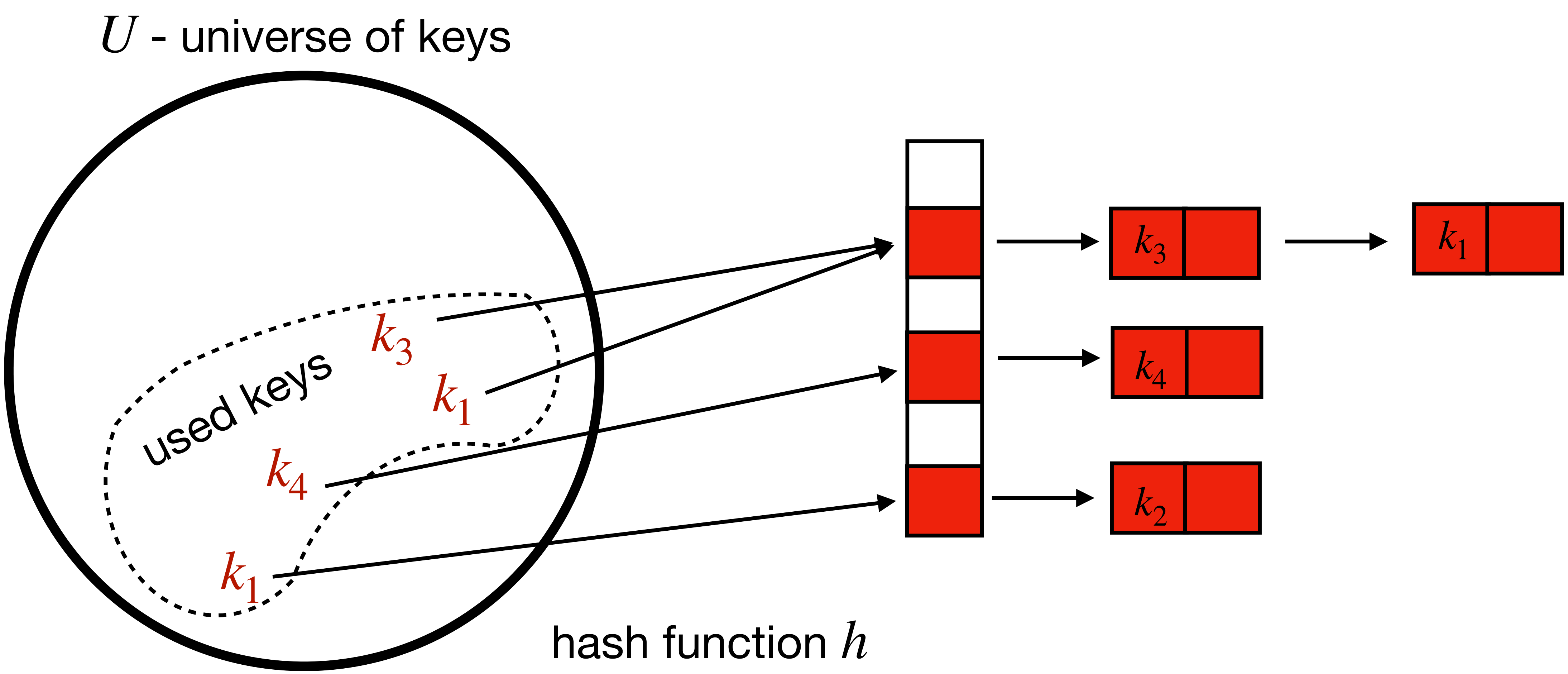
- **Insert(key, value):** insert (key, value) at the head of the list $T[h(key)]$
- **Time:** $O(1)$

Chained hash tables



- **Search(key, value):** search (key, value) in the list $T[h(key)]$
- **Time:** $O(|T[h(key)]|)$

Chained hash tables



- **Delete(key, value):** delete (key, value) from the list $T[h(key)]$
- **Time:** $O(|T[h(key)]|)$

Analysis

- Time of search and delete depends on the length of the list
- The length of the list depends on how well the keys are distributed by the hash function
- **In the worst case, search and delete cost $O(n)$ time, where n is the number of keys used**
- However, in expectation chained hash tables demonstrate much better behaviour (if h satisfies a certain assumption)

Analysis

Sample space: Finite set S of events in which we are interested.

Probability distribution: Function $\Pr : S \to \mathbb{R}$ such that $0 \leq \Pr[s] \leq 1$ for all $s \in S$ and $\sum_{s \in S} \Pr[s] = 1$.

Random variable: A real valued function of S. That is, a function $X : S \to \mathbb{R}$.

Expected value of a random variable: The expected value of a random variable X is

$$\mathrm{E}[X] = \sum_{s \in S} \Pr(\mathrm{s}) \cdot X(s).$$

Analysis

We assume to be given a probability distribution on the universe U of keys.

This induces a distribution on the n-tuples of keys.

We want to upper-bound

$$\mathrm{E}[T_{search}(n)] = \sum_{u_1,\ldots,u_n \in U} (\text{worst-case search time for } u_1, \ldots, u_n) \cdot \Pr[k_1 = u_1, \ldots, k_n = u_n]$$

Analysis

Simple Uniform Hashing Assumption (SUHA):

"h equally distributes the keys into the table slots"

Theorem. Assuming **SUHA** and that $h(x)$ can be computed in $O(1)$ time, $\mathrm{E}[T_{search}(n)] = O(1 + n/|T|)$.

Analysis

Simple Uniform Hashing Assumption (SUHA):

$\forall y \in \{1,2,\ldots,|T|\}$, there is $\Pr[h(x) = y] = 1/|T|$, and

$\forall y_1, y_2 \in \{1,2,\ldots,|T|\}$, there is $\Pr[h(x_1) = y_1, h(x_2) = y_2] = (1/|T|)^2$

Theorem. Under **SUHA** and assuming that $h(x)$ can be computed in $O(1)$ time, $\mathrm{E}[T_{search}(n)] = O(1 + n/|T|)$.

Analysis

Suppose that a random variable X can only have values $0,1,2,\ldots,t$.

Notation: For each i, define $\Pr[X=i] = \sum_{s\in S, X(s)=i} \Pr[s]$

Claim. If the only possible values for X are $0,1,2,\ldots,t$ then

$$\mathrm{E}[X] = \sum_{i=0}^{t} i \cdot \Pr[X=i]$$

Analysis

Claim. If a random variable X is a sum of t other random variables, $X = X_1 + X_2 + \ldots + X_t$, then

$$\mathrm{E}[X] = \mathrm{E}[X_1 + X_2 + \ldots + X_t] = E[X_1] + E[X_2] + \ldots + E[X_t]$$

Application: We can find the expected value of X by finding the expected values of each of $X_1, X_2, \ldots, X_t$ and then adding these together.

Analysis

1. Unsuccessful search

Suppose that: $k_1, k_2, \ldots, k_n$ are keys in the dictionary, and we perform an unsuccessful search for a key k.

If we do not include comparisons to the null pointer, then the number of comparisons for an unsuccessful search for k is

$$X_1 + X_2 + \ldots + X_n$$

where $X_i = \begin{cases} 1 & \text{, if } h(k) = h(k_i); \\ 0 & \text{, otherwise .} \end{cases}$

The expected time is $\mathrm{E}[X] = \sum_i \mathrm{E}[X_i] = \sum_i \Pr[X_i = 1] = n/|T|$ by SUHA.

Analysis

2. Successful search

Suppose keys were introduced in order $k_1, k_2, \ldots, k_n$.

Consider a successful search for k_i.

k_i appears before any of $k_1, k_2, \ldots, k_{i-1}$ that are in the same linked list, and after any of $k_{i+1}, k_{i+2}, \ldots, k_n$ that are in the same linked list.

Analysis

2. Successful search

Number of comparisons to search for k_i is, therefore,

$$Y_i = 1 + X_{i+1} + X_{i+2} + \ldots + X_n$$

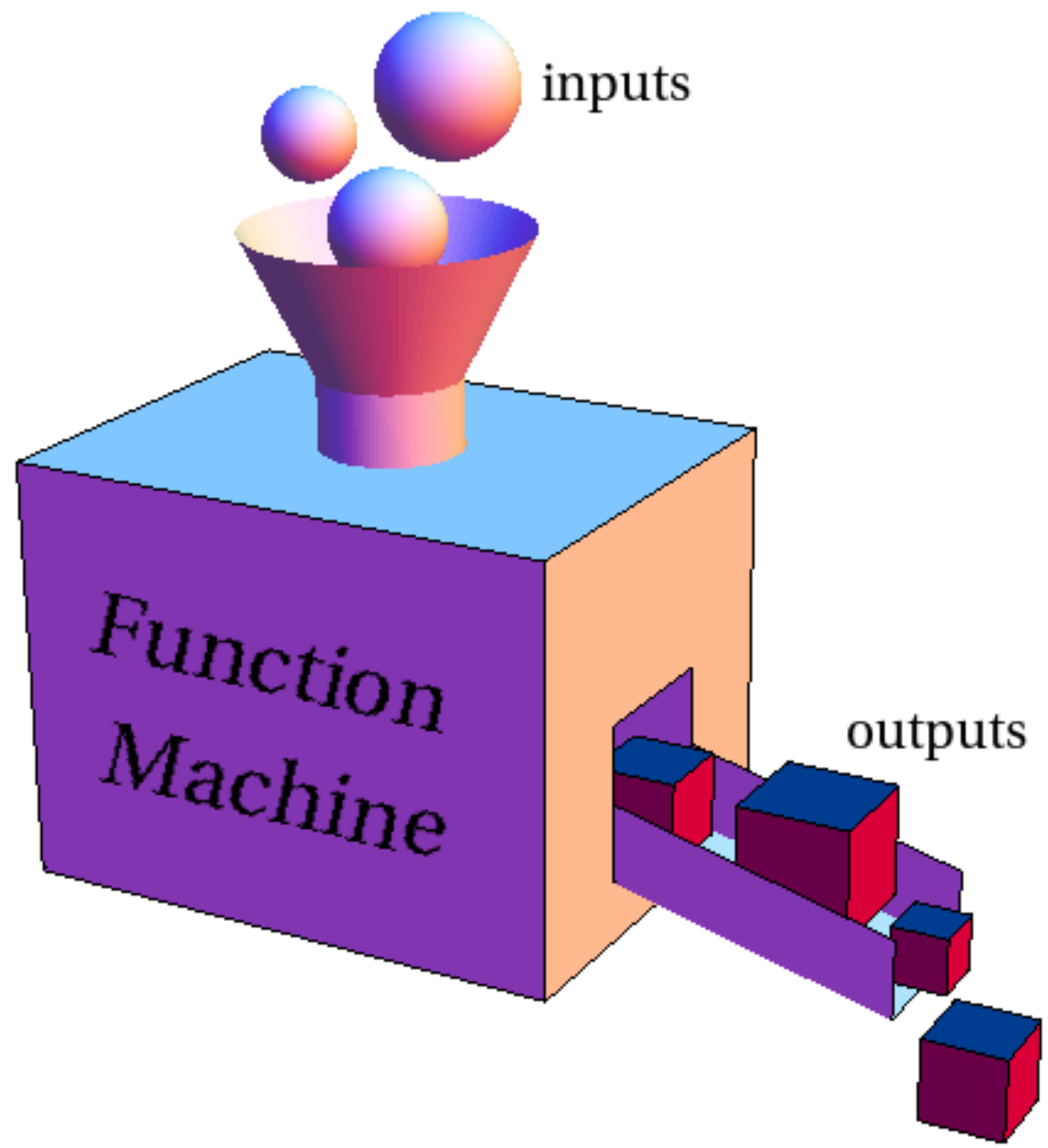
where $X_j = \begin{cases} 1 & \text{, if } h(k_j) = h(k_i); \\ 0 & \text{, otherwise}. \end{cases}$

Under SUHA, $\mathrm{E}[X_j] = 1/|T|$. We assume that each key in the table is equally likely to be searched for.

By linearity of expectations, the average expected search time is

$$\frac{1}{n}\sum_{i=1}^{n} \mathrm{E}[Y_i] = 1 + \frac{1}{n}\sum_{i=1}^{n} (n-i)/|T| = O(1 + n/|T|)$$

Designing hash functions



Designing hash functions

- Good hash functions must distribute the keys evenly
- If we do not know the distribution of the keys, it can be hard to achieve
- In practice, various heuristics are used, and we will consider several of them
- We can assume that keys are integers

Heuristic hash functions

1. **Division method:** $h(k) = k \pmod{m}$. It is better to choose m to be a prime number, and avoid $m = 2^p$ (as for this value of m the function always returns the p least significant bits of the keys)

2. **Multiplication method:** $h(k) = \lfloor m \cdot (k \cdot A \pmod{1}) \rfloor$, where $kA \pmod{1} := k \cdot A - \lfloor k \cdot A \rfloor$.

   $A \in (0,1)$, and usually, m is chosen to be 2^p.

Universal hashing

If we fix the hash function h, an adversary can always find a probability distribution on the universe of keys for which our function will be "bad"

Let $H = \{h : U \to [0,m-1]\}$ be a finite family of hash functions. It is called **universal** if

$$\forall k_1 \neq k_2 \in U, |\{h \in H : h(k_1) = h(k_2)\}| \leq |H|/m$$

In other words: if we choose $h \in H$ at random, the probability of collision for the keys k_1, k_2 is at most $1/m$.

Universal hashing

Let $H = \{h : U \to [0,m-1]\}$ be a finite family of hash functions. It is called **universal** if

$$\forall k_1 \neq k_2 \in U, |\{h \in H : h(k_1) = h(k_2)\}| \leq |H|/m$$

Theorem. Let h be a hash function chosen uniformly at random from a universal family of hash functions. Suppose that $h(k)$ can be computed in constant time and that there are n keys. Then the expected search time for hashing with chaining is $O(1 + n/m)$.

Proof: Analogous to the case when h satisfies SUHA.

Universal hashing

Let $H = \{h : U \to [0,m-1]\}$ be a finite family of hash functions. It is called **universal** if

$$\forall k_1 \neq k_2 \in U, |\{h \in H : h(k_1) = h(k_2)\}| \leq |H|/m$$

Theorem. Let h be a hash function chosen uniformly at random from a universal family of hash functions. Suppo be computed in constant time and that there are expected search time for hashing with chaining is

But the distribution is over the hash functions, not keys!

Proof: Analogous to the case when h satisfies SUHA.

Universal hashing

We will now construct a universal family of hash functions.

Let p be a prime number such that $[0,p-1] \supseteq U$. Define

$$H = \{h_{a,b}(k) = ((ak + b) \mod p) \mod m : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

Theorem. H is a universal family of hash functions.

Universal hashing

$H = \{h_{a,b}(k) = ((ak + b) \mod p) \mod m : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$

Theorem. H is a universal family of hash functions.

Fix $k_1 \neq k_2 \in U$. Let $\ell_1 = (ak_1 + b) \mod p$ and $\ell_2 = (ak_2 + b) \mod p$. We have $\ell_1 \neq \ell_2$. The number of such pairs is $p(p-1)$. Moreover,

$$a = ((\ell_1 - \ell_2)((k_1 - k_2)^{-1} \mod p) \mod p)$$
$$b = (\ell_1 - ak_1) \mod p$$

Hence, there is one-to-one mapping between (a, b) and (ℓ_1, ℓ_2).

Universal hashing

$$H = \{h_{a,b}(k) = ((ak + b) \mod p) \mod m : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

Theorem. H is a universal family of hash functions.

The number of $h \in H$ such that $h(k_1) = h(k_2)$ is equal to

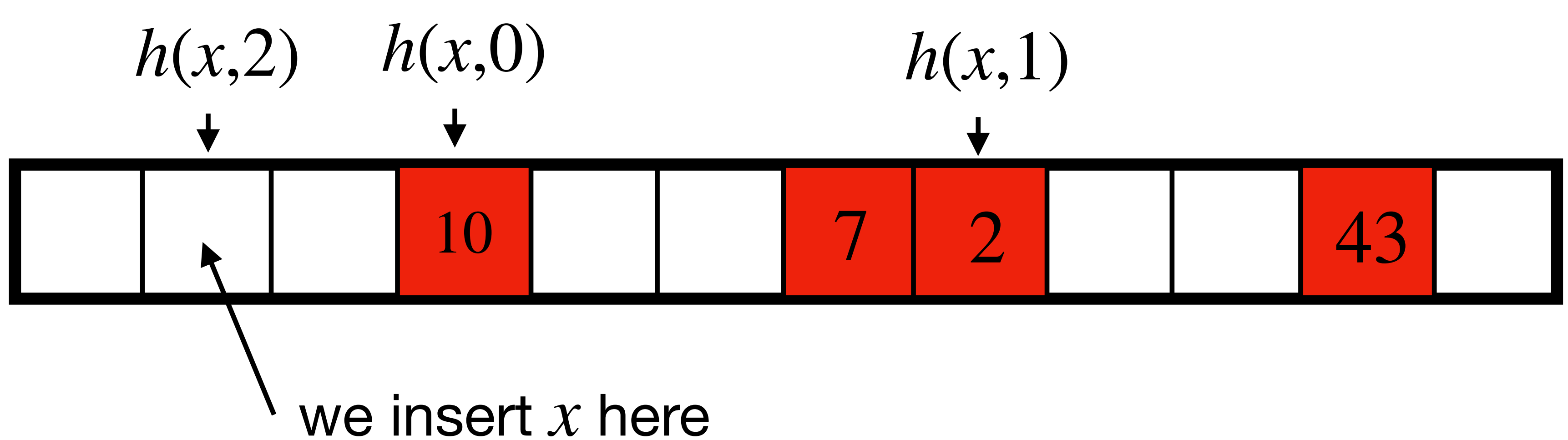
$$|\{(\ell_1, \ell_2) : \ell_1 \neq \ell_2 \in \mathbb{Z}_p, \ell_1 = \ell_2 \pmod{m}\}| \leq p(p-1)/m = |H|/m$$

q.e.d.

Open addressing

Open addressing

- Elements are stored in the table
- Insertion(x): **probe** the hash table until we find x or an empty slot. If we find an empty slot, insert x there.
- To define which slots to probe, we use a hash function that depends **on the key and the probe number**

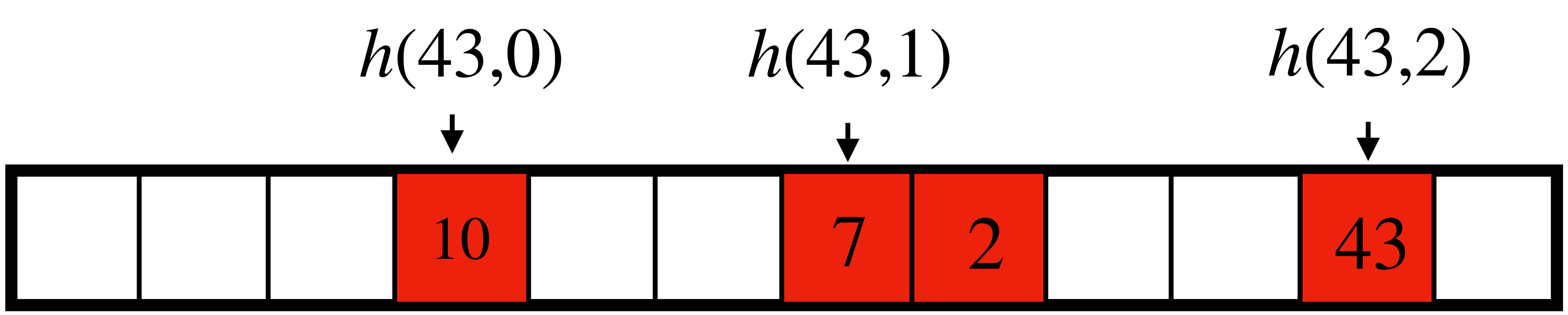


Open addressing

- Elements are stored in the table

- Search(x): **probe** the hash table until either we find x (return YES) or an empty slot (return NO)

- To define which slots to probe, we use a hash function that depends **on the key and the probe number**

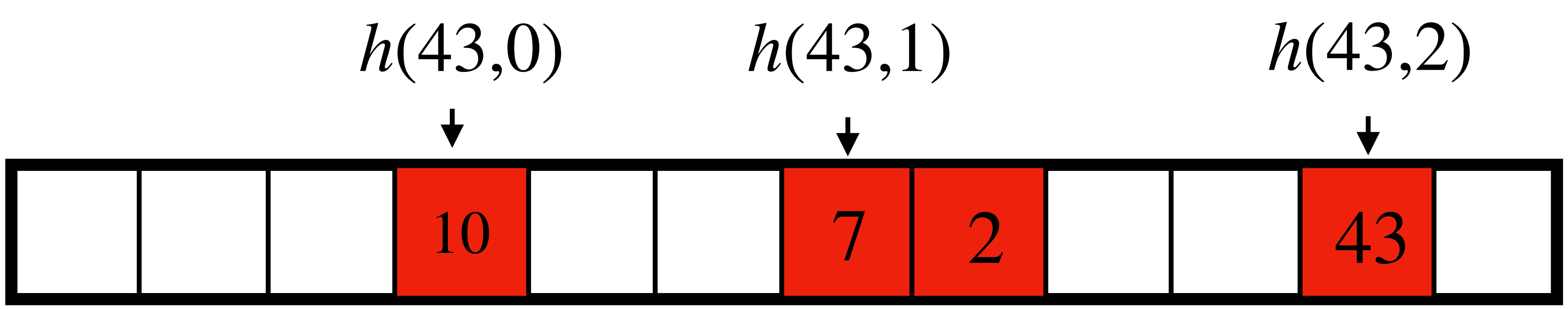


Which hash function to use?

- In the analysis, we will assume that h is uniform, i.e. the probe sequence of a key is equally likely to be any of the $m!$ permutations of the slots

- Gives good results, but hard to implement

- In practice, it's common to use heuristics



Heuristic hash functions

$h', h'' : U \to \{0,1,\ldots,m-1\}$ - auxiliary hash functions

Linear probing: $h(k,i) = (h'(k) + i) \mod m$

Easy to implement, but suffers from clustering

Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$

Must choose the constants c_1, c_2 carefully, suffers from clustering as well

Double hashing: $h(k,i) = (h'(k) + ih''(k)) \mod m$

To use the whole table, $h''(k)$ must be relatively prime to m, e.g. $h''(k)$ is always odd, $m = 2^i$.

Analysis of open-address hashing

Theorem. Given an open-address hash-table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Unsuccessful search(x) = every probed slot except the last one is occupied and does not contain x; the last slot is empty.

A_i - ith probe occurs and the slot is occupied

$\Pr[\text{\# of probes} \geq i] = \Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}] =$

$= \Pr[A_1] \cdot \Pr[A_2 | A_1] \cdot \Pr[A_3 | A_1 \cap A_2] \ldots \Pr[A_{i-1} | A_1 \cap A_2 \cap \ldots \cap A_{i-2}]$

(Reminder: $Pr[A \mid B] := \dfrac{Pr[A \cap B]}{Pr[B]}$, the proof is by induction)

Analysis of open-address hashing

Theorem. Given an open-address hash-table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

We must estimate

$\Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \ldots \Pr[A_{i-1} \mid A_1 \cap A_2 \cap \ldots \cap A_{i-2}]$

We have:

$\Pr[A_1] = n/m$ (n cells out of m are occupied)

$\Pr[A_2 \mid A_1] = (n-1)/(m-1)$ (we can hit one of the remaining $n-1$ elements in $m-1$ cells)

…

$\Pr[A_{i-1} \mid A_1 \cap A_2 \cap \ldots \cap A_{i-2}]$ = (n-i+2)/(m-i+2)

Analysis of open-address hashing

Theorem. Given an open-address hash-table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

We must estimate

$$\Pr[A_1] \cdot \Pr[A_2 \,|\, A_1] \cdot \Pr[A_3 \,|\, A_1 \cap A_2] \ldots \Pr[A_{i-1} \,|\, A_1 \cap A_2 \cap \ldots \cap A_{i-2}] \leq$$

$$\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdot \ldots \cdot \frac{n-i+2}{m-i+2} \leq (\frac{n}{m})^{i-1} = \alpha^{i-1}$$

Expected number of probes =

$$\sum_{i=1}^{\infty} \Pr[\text{\# of probes} \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

Analysis of open-address hashing

Corollary. The expected number of probes during insertion(x) is at most $1/(1-\alpha)$, assuming uniform hashing.

If we insert x, we first run an unsuccessful search for it.

Analysis of open-address hashing

Theorem. The expected number of probes during a successful search is at most $\dfrac{1}{\alpha}\ln\dfrac{1}{1-\alpha}$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

A successful search for x probes the same sequence of slots as insertion(x).

If x is the i-th item inserted into the table, insertion(x) probes $\leq 1/(1-i/m)$ slots in expectation.

Therefore, the expected time of a successful search is at most

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \overset{(*)}{\leq} \frac{m}{n}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

$$H_n = \sum_{i=1}^{n}\frac{1}{n} = \ln n + \gamma + \frac{1}{2n} - \varepsilon_n,$$

where $\gamma \approx 0.5772$ is the Euleur-Mascheroni constant and $0 \leq \varepsilon_n \leq 1/(8n^2)$

Cuckoo hashing

Cuckoo hashing

- In hashing with chaining and open-address hashing, the search time is good only on average.
- Can we design a hashing scheme with search time constant in the worst case?
- **Perfect hashing:** maintain a hash function that has no collisions for elements in the set. This allows to insert the elements directly into the array, without having to use a linked list. If the set of the keys is static, such functions do exist, but the construction is rather complicated.
- We will consider a simple scheme called cuckoo hashing from the 2004 paper of Pagh and Rodler (available on Moodle).

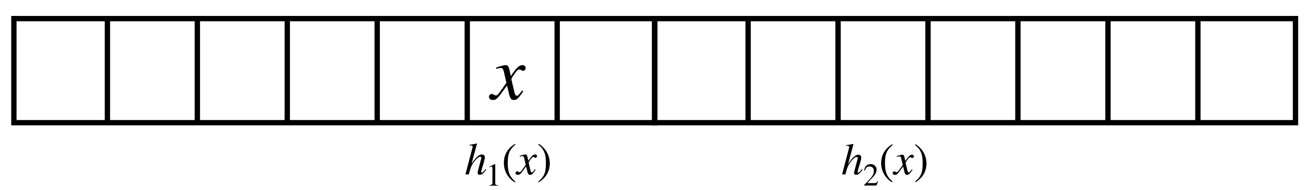
Cuckoo hashing

Assume that $h_1, h_2 : U \to [1, |T|]$ satisfy SUHA:

$\forall y \in \{1,2,\ldots,|T|\}$, there is $\Pr[h_i(x) = y] = 1/|T|$, and

$\forall y_1, y_2 \in \{1,2,\ldots,|T|\}$, there is $\Pr[h_i(x_1) = y_1, h_i(x_2) = y_2] = (1/|T|)^2$

table T



Invariant: we store an element x either in $T[h_1(x)]$, or in $T[h_2(x)]$

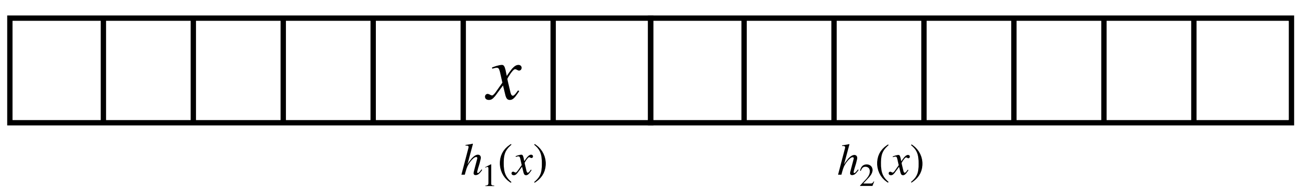
Search for x: compare x with $T[h_1(x)]$ and $T[h_2(x)]$ **- constant time!**

Cuckoo hashing

Insertion

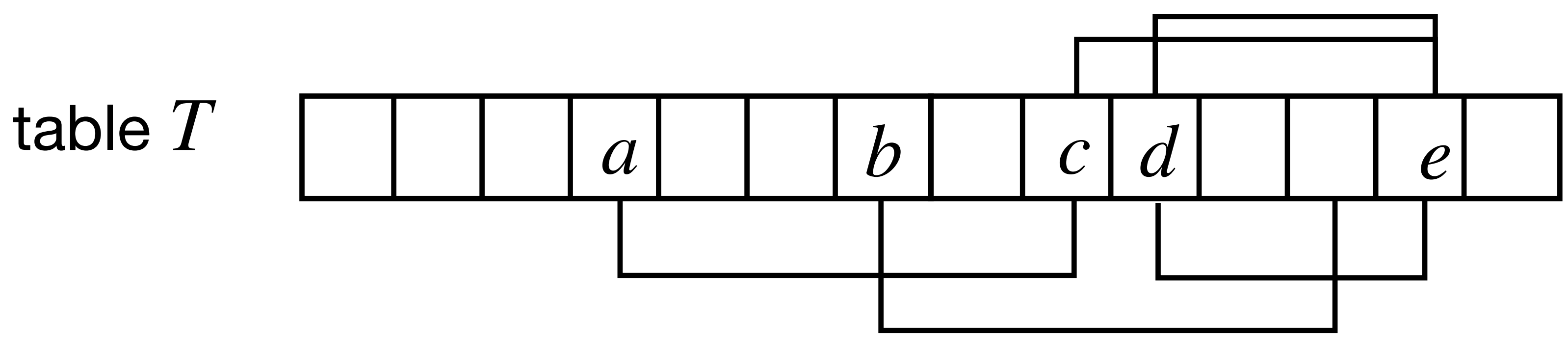
Main idea. Try to put x into $T[h_1(x)]$. If $T[h_1(x)]$ is empty, we are done. Otherwise replace $y = T[h_1(x)]$ with x [as a cuckoo does], and repeat for y.

table T




```
Insert(x)
if x = T[h1(x)] or x = T[h2(x)] then return
pos ← h1(x)
loop n times:
    if T[pos] = NULL then T[pos] = x; return
    x ↔ T[pos]  \\swap the contents of x and T[pos]
    if pos = h1(x) then pos ← h2(x) else pos ← h1(x)
rehash \\choose new h1, h2 and insert all elements from scratch
Insert(x)
```

Cuckoo hashing

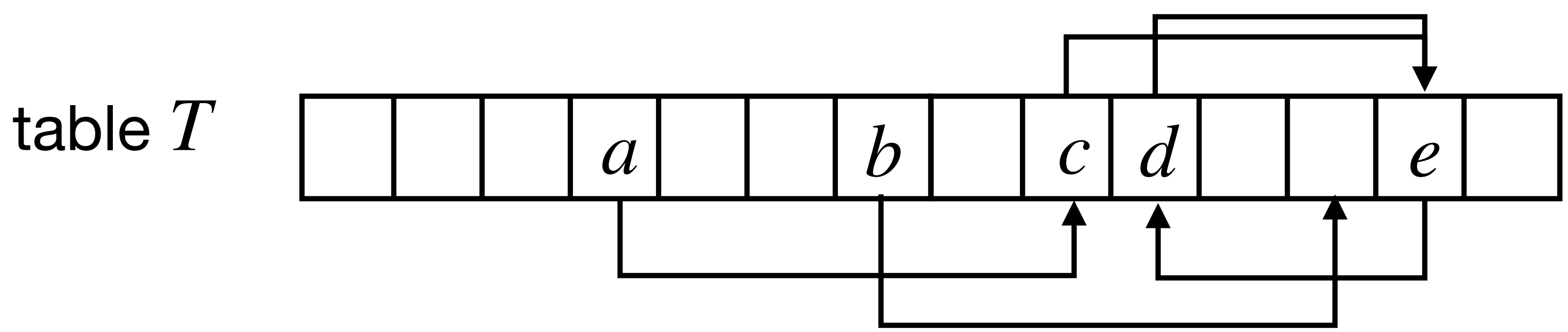


<u>Lemma.</u> Suppose that $|T| \geq c \cdot n$ for some constant $c > 1$. For any i, j the probability that there exists a path from i to j of length $\ell \geq 1$, which is a shortest path from i to j, is at most $1/(c^{\ell} \cdot |T|)$.

<u>Proof.</u> By induction on ℓ. Base case $\ell = 1$. By SUHA, $\Pr[h_{1/2}(x) = y] = 1/|T|$. Therefore,

$\Pr[$there is an edge from i to $j] = n/|T|^2 \leq 1/(c \cdot |T|)$

Cuckoo hashing

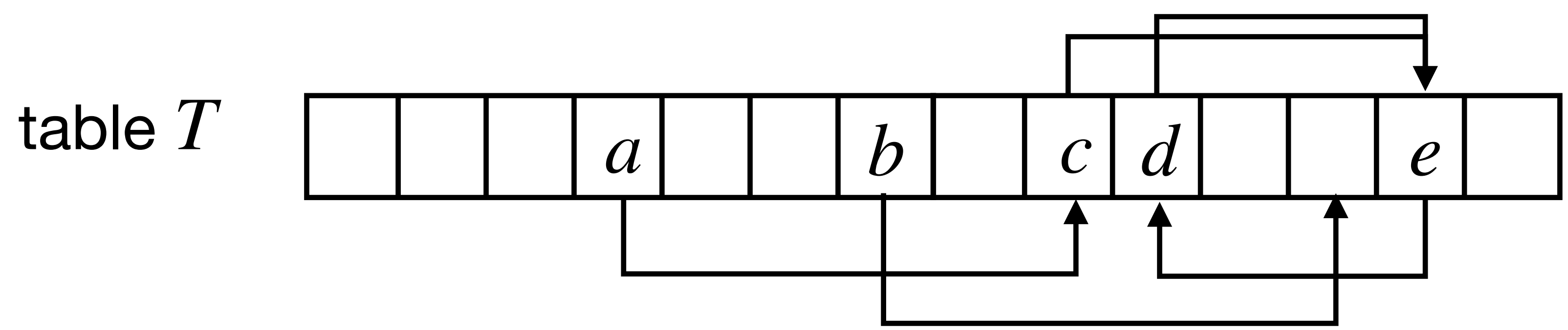


Lemma. Suppose that $|T| \geq c \cdot n$ for some constant $c > 9$. For any i, j the probability that there exists a path from i to j of length $\ell \geq 1$, which is a shortest path from i to j, is at most $1/(c^{\ell} \cdot |T|)$.

Proof. By induction on ℓ. For $\ell \geq 1$, there must exist k such that there is a path of length $\ell - 1$ from i to k and an edge from k to j.

$\Pr[\text{there is a path from } i \text{ to } j] = |T| \cdot (1/(c^{\ell-1} \cdot |T|)) \cdot (1/(c \cdot |T|)) \leq 1/(c^{\ell} \cdot |T|)$

Cuckoo hashing

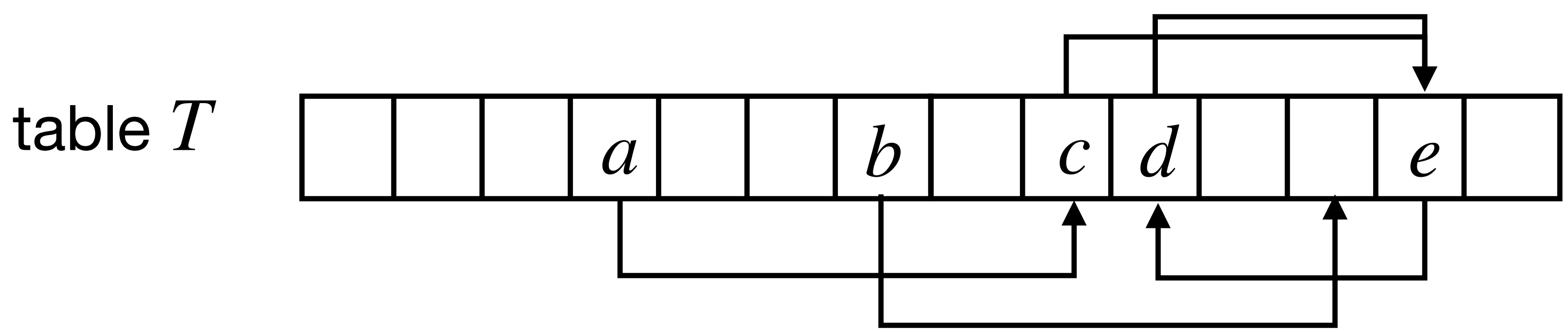


Bucket of x: all cells that can be reached either from $h_1(x)$ or $h_2(x)$

x, y are in the same bucket ⇔ there is a path from $\{h_1(x), h_2(x)\}$ to $\{h_1(y), h_2(y)\}$

$$\Pr[x, y \text{ are in the same bucket}] \leq 4 \sum_{\ell=1}^{\infty} 1/(c^{\ell} \cdot |T|) = 4/((c-1) \cdot |T|)$$

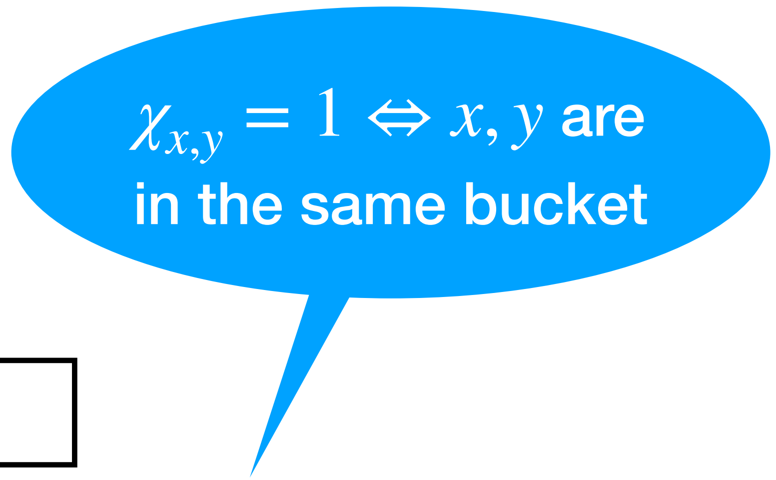
Cuckoo hashing

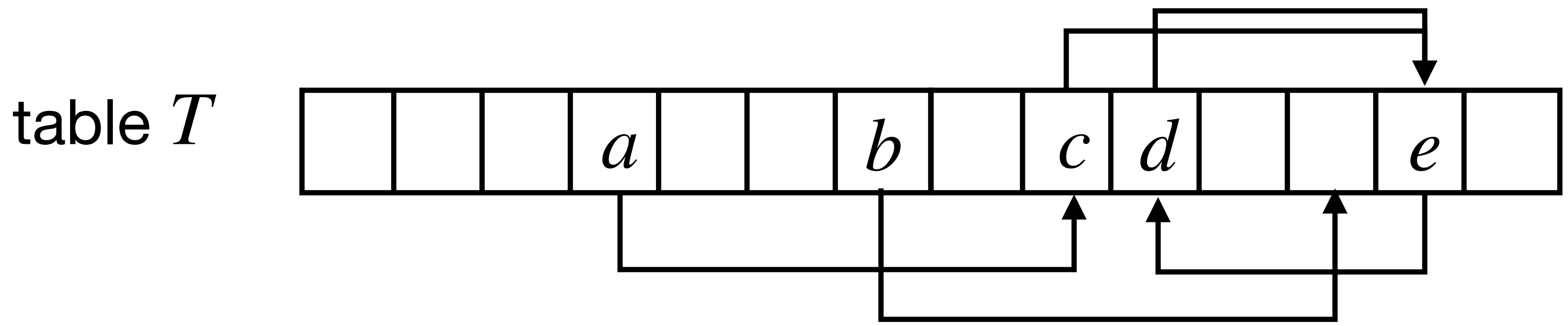


$$\Pr[x, y \text{ are in the same bucket}] \leq 4 \sum_{\ell=1}^{\infty} 1/(c^{\ell} \cdot |T|) = 4/((c-1) \cdot |T|)$$

Expected size of the bucket of x is at most $4/(c-1) \Leftrightarrow$ in the absence of rehash, expected insertion time is constant.

Cuckoo hashing





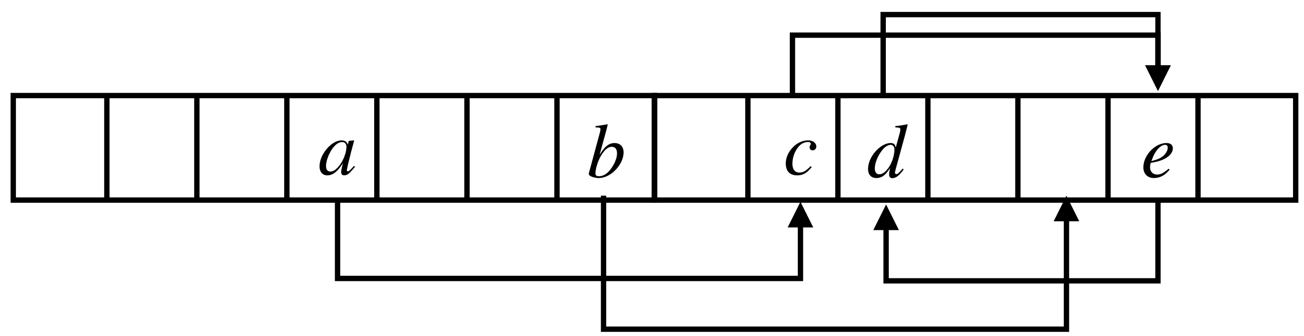
$$\Pr[x, y \text{ are in the same bucket}] \leq 4 \sum_{\ell=1}^{\infty} 1/(c^{\ell} \cdot |T|) = 4/((c-1) \cdot |T|)$$

$$\mathbb{E}[\,|\text{bucket of } x|\,] = \mathbb{E}[\sum \chi_{x,y}] = \sum \mathbb{E}[\chi_{x,y}] = \sum 1 \cdot \Pr[x, y \text{ are in the same bucket}] \leq 4/(c-1)$$

Hence, in the absence of rehash, expected insertion time is constant.

Cuckoo hashing

table T



Probability that we need a rehash is at most probability that there is a cycle, i.e. there is a path from i to itself: $\dfrac{4}{c-1} \leq 1/2$

Probability that we will need two rehashes is at most $1/2^2$ … and so on

Hence, expected time per insertion: $\dfrac{1}{n} \cdot O(n) \cdot \sum_{i=1}^{\infty} \dfrac{1}{2^i} = O(1)$

Rolling hash functions

Pattern matching

text

0	1	1	0	1	0	0	0	1	1	0	1	0	1

pattern

1	1	0	1	0

Input: a pattern (a string of length m) and a text (a string of length n)

Output: all occurrences of the pattern in the text

Karp-Rabin fingerprint

The Karp-Rabin fingerprint of a string $S = s_1 s_2 \ldots s_m$ is defined as

$$\varphi(s_1 s_2 \ldots s_m) = \sum_{i=1}^{m} s_i \cdot r^{m-i} \bmod p,$$

where p is a prime and r is a random integer in $\mathbb{F}_p$.

It's a good hash function:

- If $S = T$, then $\varphi(S) = \varphi(T)$;
- If $S \neq T$ while the lengths of S and T are equal, then $\varphi(S) \neq \varphi(T)$ with high probability (if p is large enough).

Let's zoom in...

Karp-Rabin fingerprint

Let $S = s_1 s_2 \ldots s_m$, $T = t_1 t_2 \ldots t_m$, and σ be the size of the alphabet. Let $p \geq \max\{\sigma, n^c\}$, where $c > 1$ is a constant.

$$\varphi(S) = \varphi(T) \Leftrightarrow \sum_{i=1}^{m} (s_i - t_i) \cdot r^{m-i} \bmod p = 0$$

Hence, r is a root of $P(x) = \sum_{i=1}^{m} (s_i - t_i) \cdot x^{m-i}$, a polynomial over $\mathbb{F}_p$. The number of roots of this polynomial is at most m.

The probability of such event is at most $m/p \leq 1/n^{c-1}$.

Karp-Rabin algorithm

- Compute the fingerprint of the pattern.

- Compare it with the fingerprint of each m-length substring of the text. If the fingerprint of the pattern is equal to the fingerprint of a substring, report it as an occurrence.

- The algorithm **never misses an occurrence**!

- It can say that a substring is an occurrence of the pattern when it is not, but only with probability at most $1/n^{c-1}$

Karp-Rabin algorithm

- Compute the fingerprint of the pattern.
- Compare it with the fingerprint of each m-length substring of the text. If the fingerprint of the pattern is equal to the fingerprint of a substring, report it as an occurrence.
- The algorithm **never misses an occurrence**!
- It can say that a substring is an occurrence of the pattern when it is not, but only with probability at most $1/n^{c-1}$

Karp-Rabin algorithm

How to compute the fingerprints?

$$\varphi(s_1 s_2 \ldots s_j) = \sum_{i=1}^{j} s_i \cdot r^{j-i} \bmod p$$

$$\varphi(s_1 s_2 \ldots s_{j+1}) = \sum_{i=1}^{j+1} s_i \cdot r^{j+1-i} \bmod p$$

Therefore, $\varphi(s_1 s_2 \ldots s_{j+1}) = \varphi(s_1 s_2 \ldots s_j) \cdot r + s_{j+1} \mod p$.

Hence, we can compute the Karp-Rabin fingerprint of the first m-length substring of the text using $O(1)$ space and $O(1)$ time per letter.

Karp-Rabin algorithm

How to compute the fingerprints?

$$\varphi(s_1 s_2 \dots s_m) = \sum_{i=1}^{m} s_i \cdot r^{m-i} \bmod p$$

$$\varphi(s_2 \dots s_{m+1}) = \sum_{i=1}^{m} s_{i+1} \cdot r^{m-i} \bmod p$$

this is why it's "rolling"!

Therefore,

$$\varphi(s_2 \dots s_{m+1}) = (\varphi(s_1 s_2 \dots s_m) - s_1 \cdot r^{m-1}) \cdot r + s_{m+1} \mod p.$$

Hence, we can compute the fingerprint of the $(i+1)$-th m-length substring of the text from the fingerprint of the i-th substring in $O(1)$ space and $O(1)$ time.

Karp-Rabin algorithm

- $O(1)$ time per letter
- $O(1)$ *extra* space (all the space except for the space required to store the input)
- Reports all occurrences of the pattern (no false-negative errors)
- Has small false-positive error

State-of-the-art:

Deterministic algorithm with $O(1)$ *extra* space and $O(1)$ time per letter

Randomised algorithm with $O(\log m)$ space and $O(1)$ time per letter

Open question: a randomised algorithm with better space?

Today's lecture

- Chained hash tables
- Designing hash functions
- Open addressing
- Cuckoo hashing
- Rolling hash functions

Next lecture

- Binary search trees
- Lower bound for sorting
- Predecessor problem