Algorithms and Datastructures

Lecture 7

Manfred Jaeger

AALBORG UNIVERSITET

Hash Tables

Data: set of objects with designated *key* attribute

Operations:

Name	Specification
Object *search(Key k)*	returns an object o with $o.key = k$ if such an object exists in the set
void *insert(Object o)*	adds o to the set
void *delete(Object o)*	deletes o from the set (o an element of the set)

Keys can be

- Names, identifiers, email addresses ...
- Social security numbers, ZIP codes, ...
- Dates, Names + Dates, ...

For Hashing techniques keys need to have non-negative integer values (natural numbers).

Example: Strings

String	P	e	t	e	
ASCII val.	80	101	116	101	
Multiply ...	$\cdot 128^3$	$\cdot 128^2$	$\cdot 128^1$	$\cdot 128^0$	
Gives ...	167.772.160	1.654.784	14.848	101	
Sum:					169.441.893

- ☞ Alternative view: concatentation of binary representations of ASCII values
- ☞ Different strings are turned into different numbers (up to numerical overflow)

Java: Object.hashCode()

C#: Object.GetHashCode()

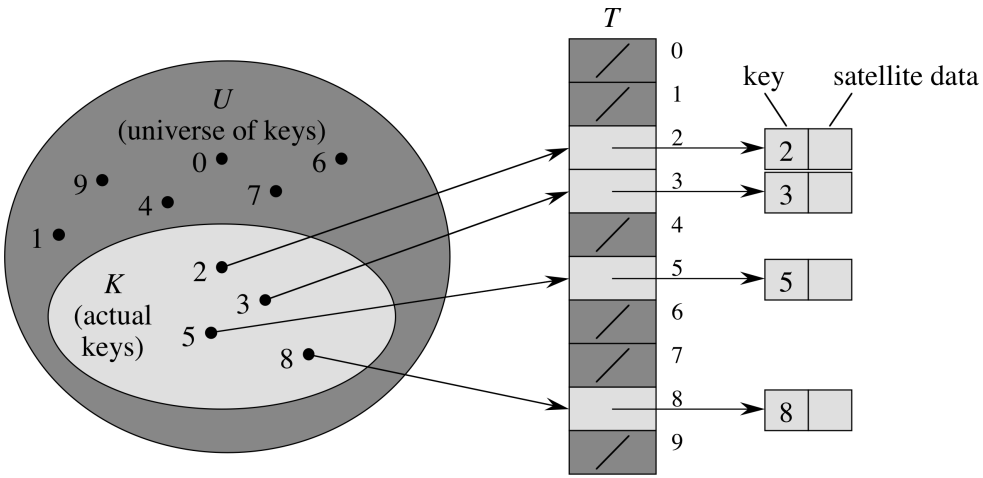
return integer values, so that

- if $o1.equals(o2)$ then $o1.hashCode() = o2.hashCode()$
- **Not:** if $o1.hashCode() = o2.hashCode()$ then $o1.equals(o2)$

From now on:

- *universe of possible keys* is
  - $U = \{0, 1, \ldots, m-1\}$ for some integer m, or
  - $U = \{0, 1, \ldots\}$ (all natural numbers)
- The set of keys of the elements stored in the dictionary is a subset $K \subseteq U$ of *actual keys*.

If $U = \{0, 1, \ldots, m-1\}$ for a small m, then a dictionary can be implemented by an array of size m:



[ItoA, Fig. 11.1]

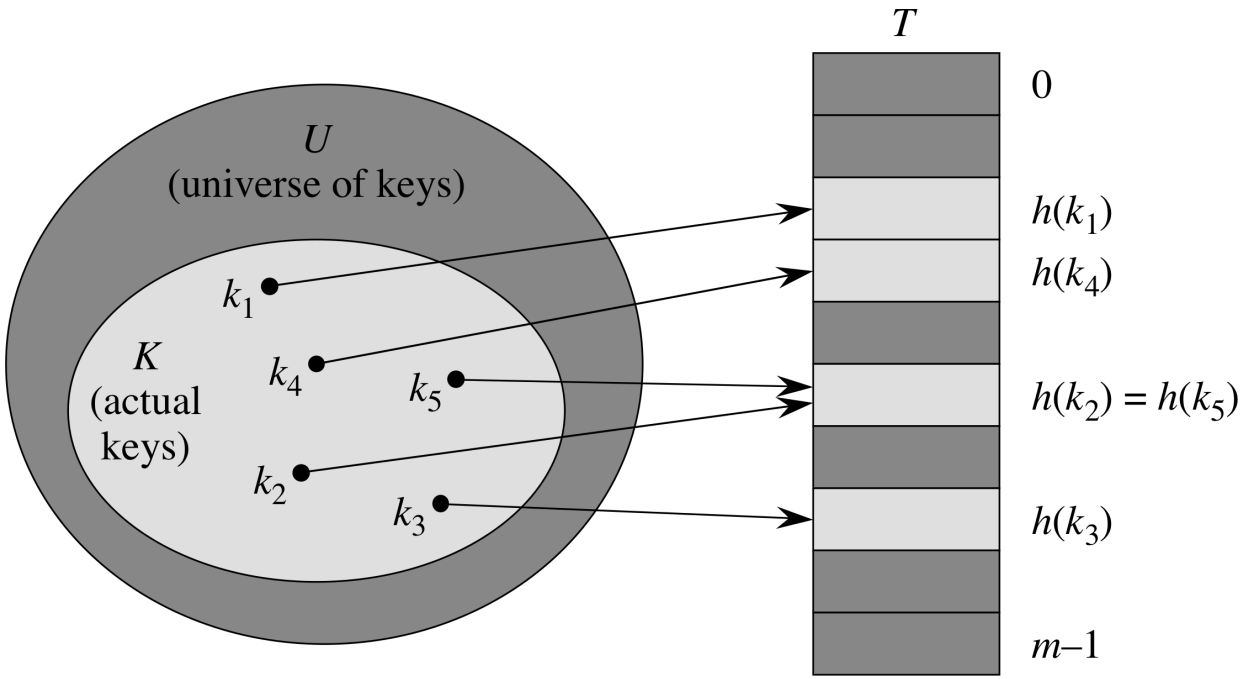
search(k): **Return** $T[k]$
insert(o): $T[o.key] = o$
delete(o): $T[o.key] = null$

☞ Assumption here: different objects have different keys

Principle: Instead of using the actual key value k, use a **hash value** $h(k)$ to determine the array index for storing k.

$$h : U \to \{0, \ldots, m-1\}$$

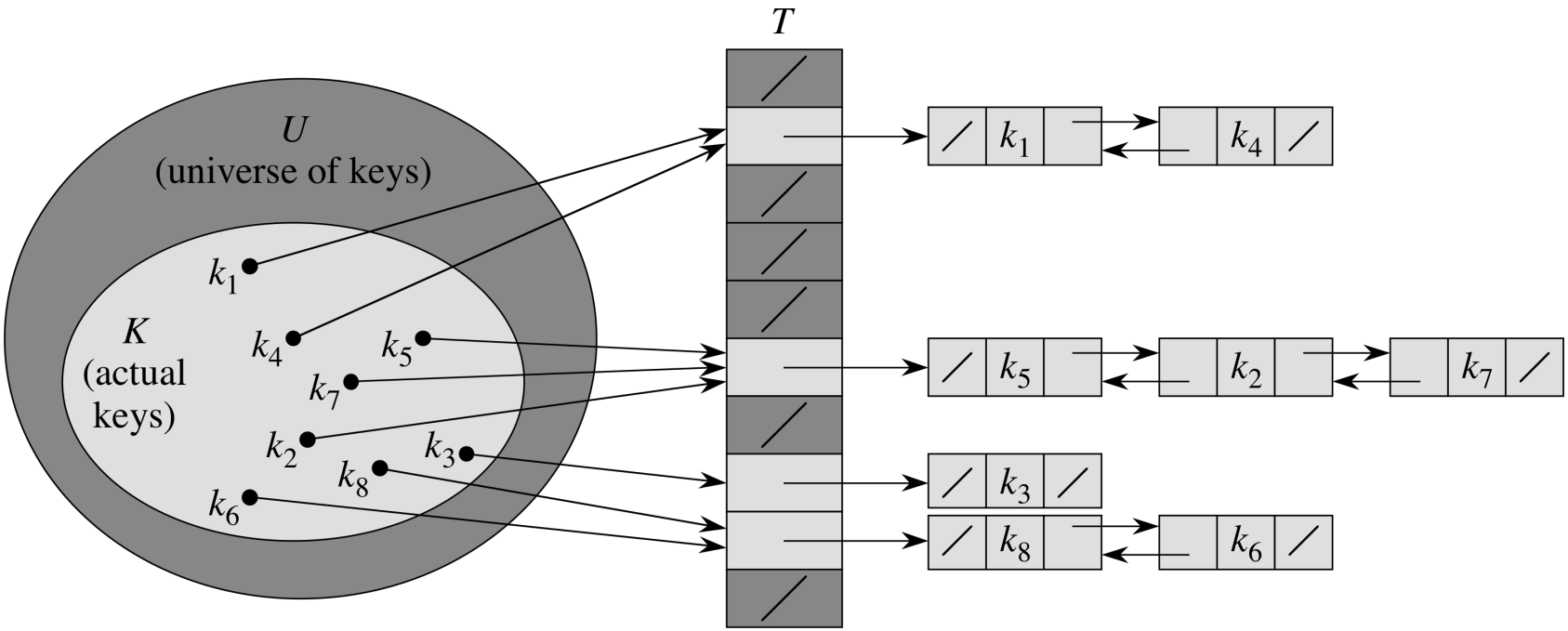
then is the **hash function**.



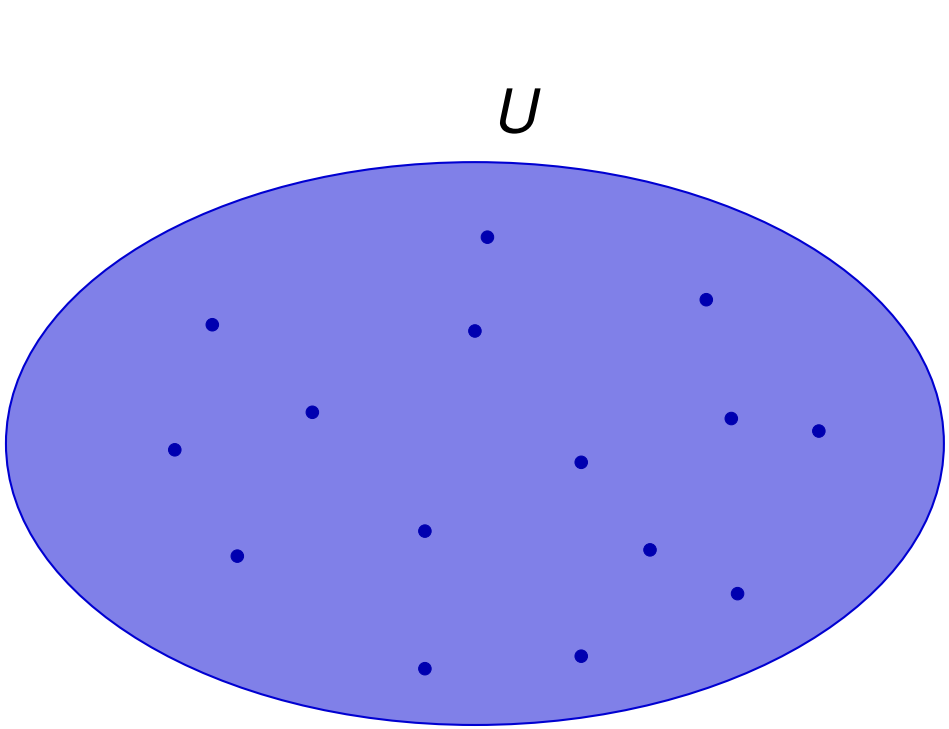
[ItoA, Fig. 11.2]

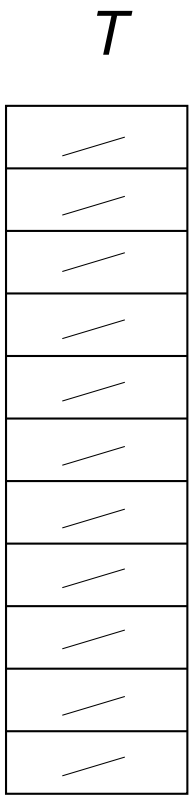
Chaining

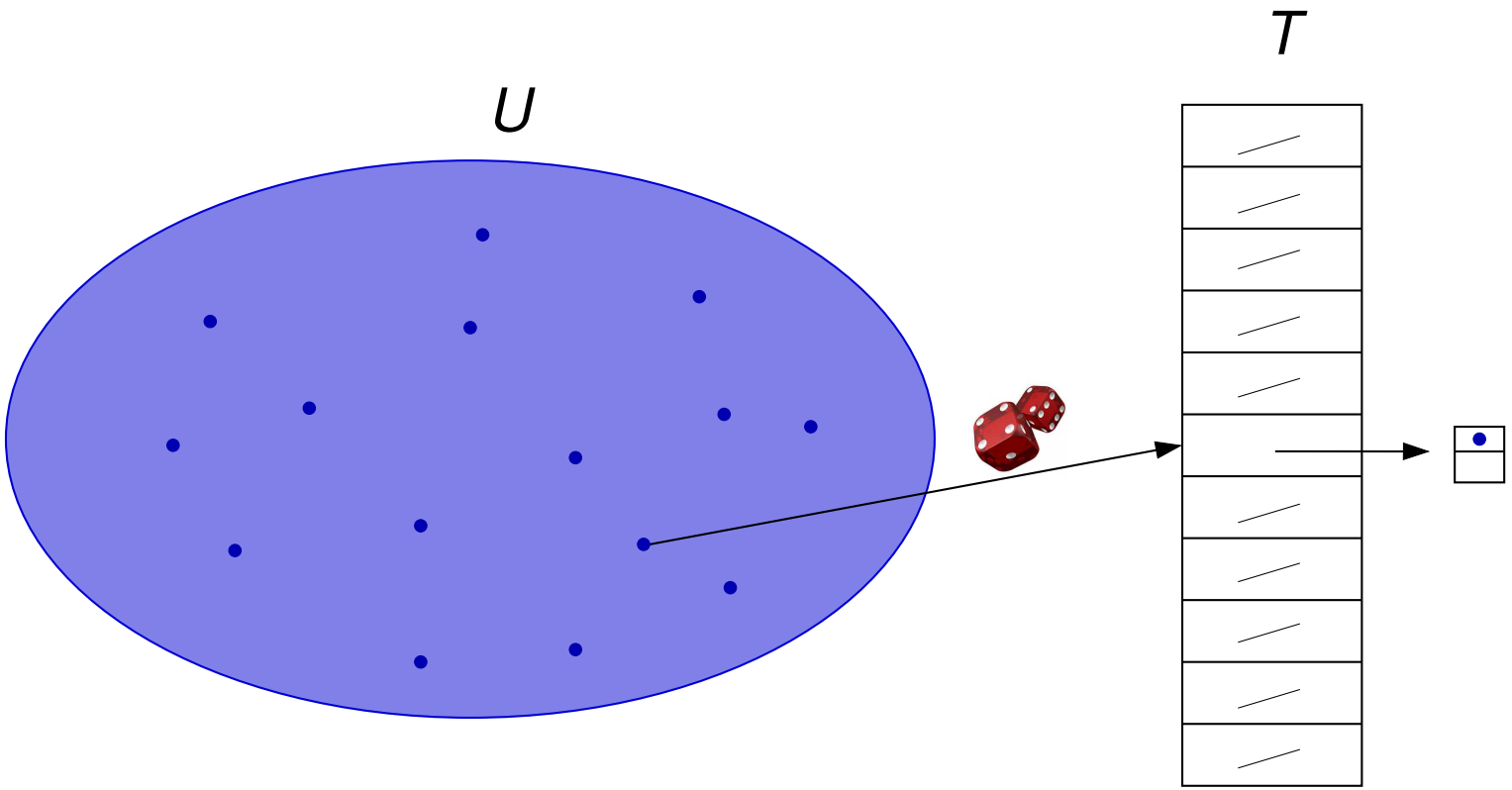
Principle: Instead of using an array of objects, use an *array of linked lists of objects*.

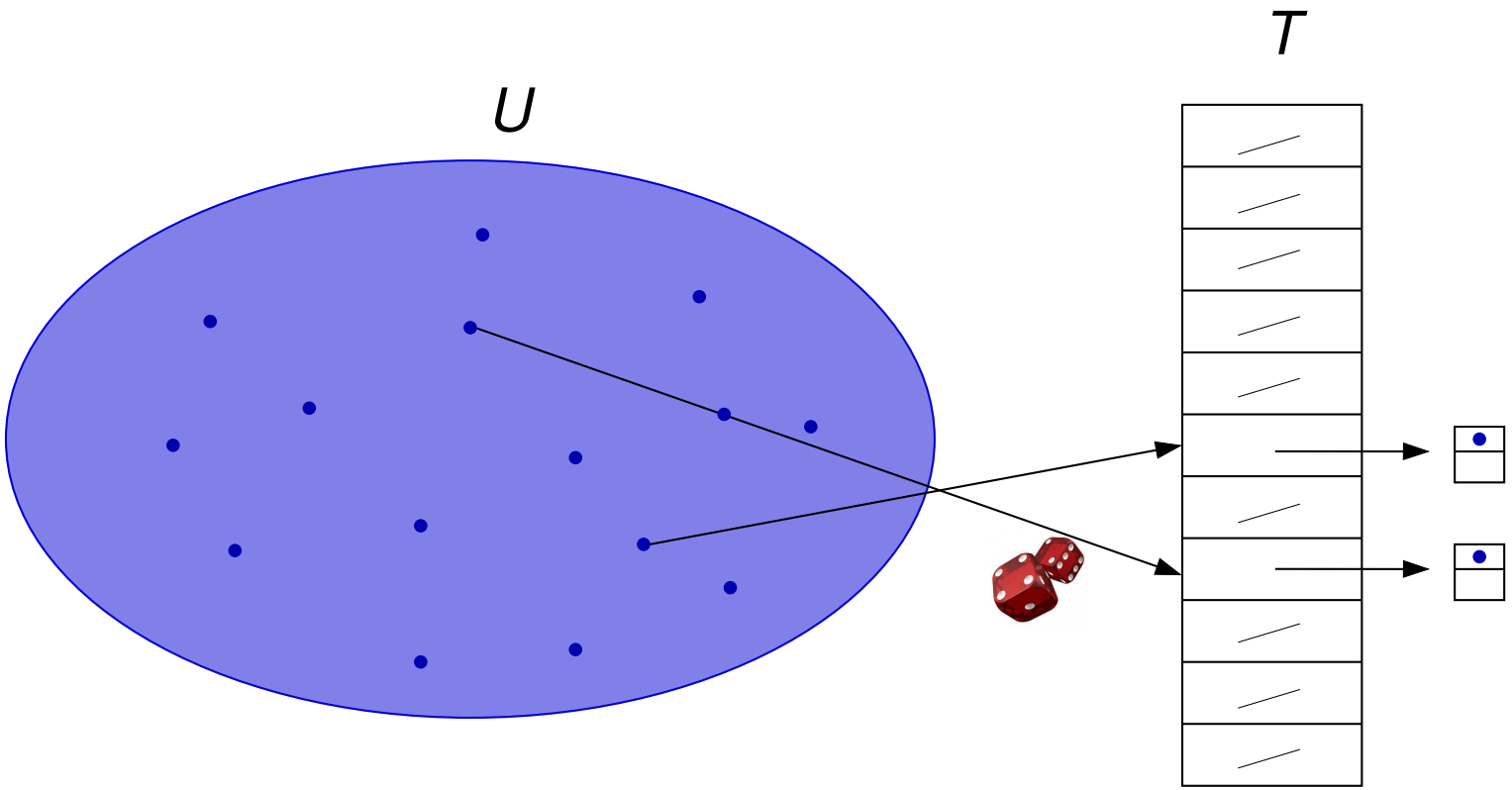


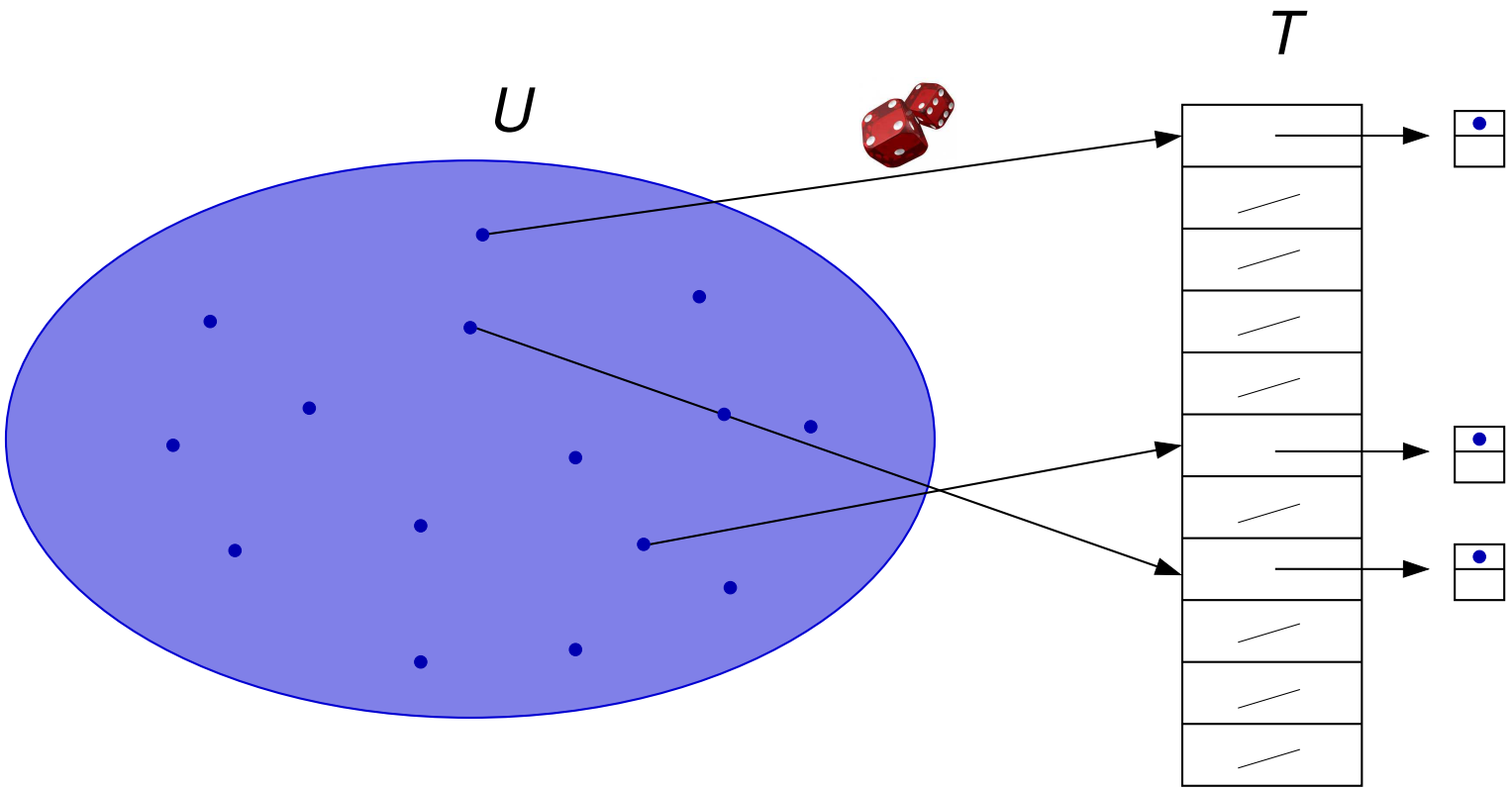
search(k): **Return** $T[h(k)].search(k)$
insert(o): $T[h(o.key)].insert(o)$
delete(o): $T[h(o.key)].delete(o)$

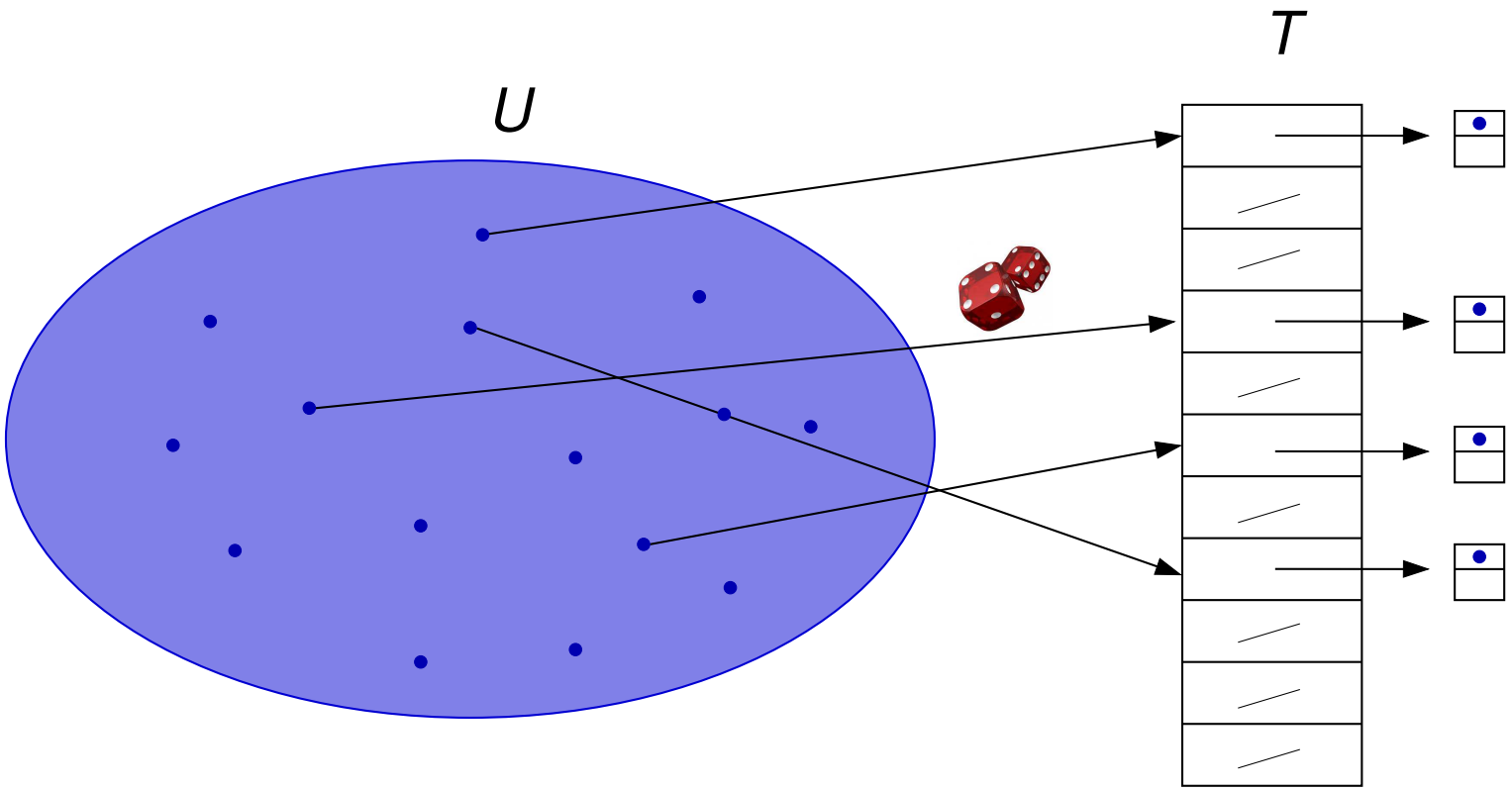

U

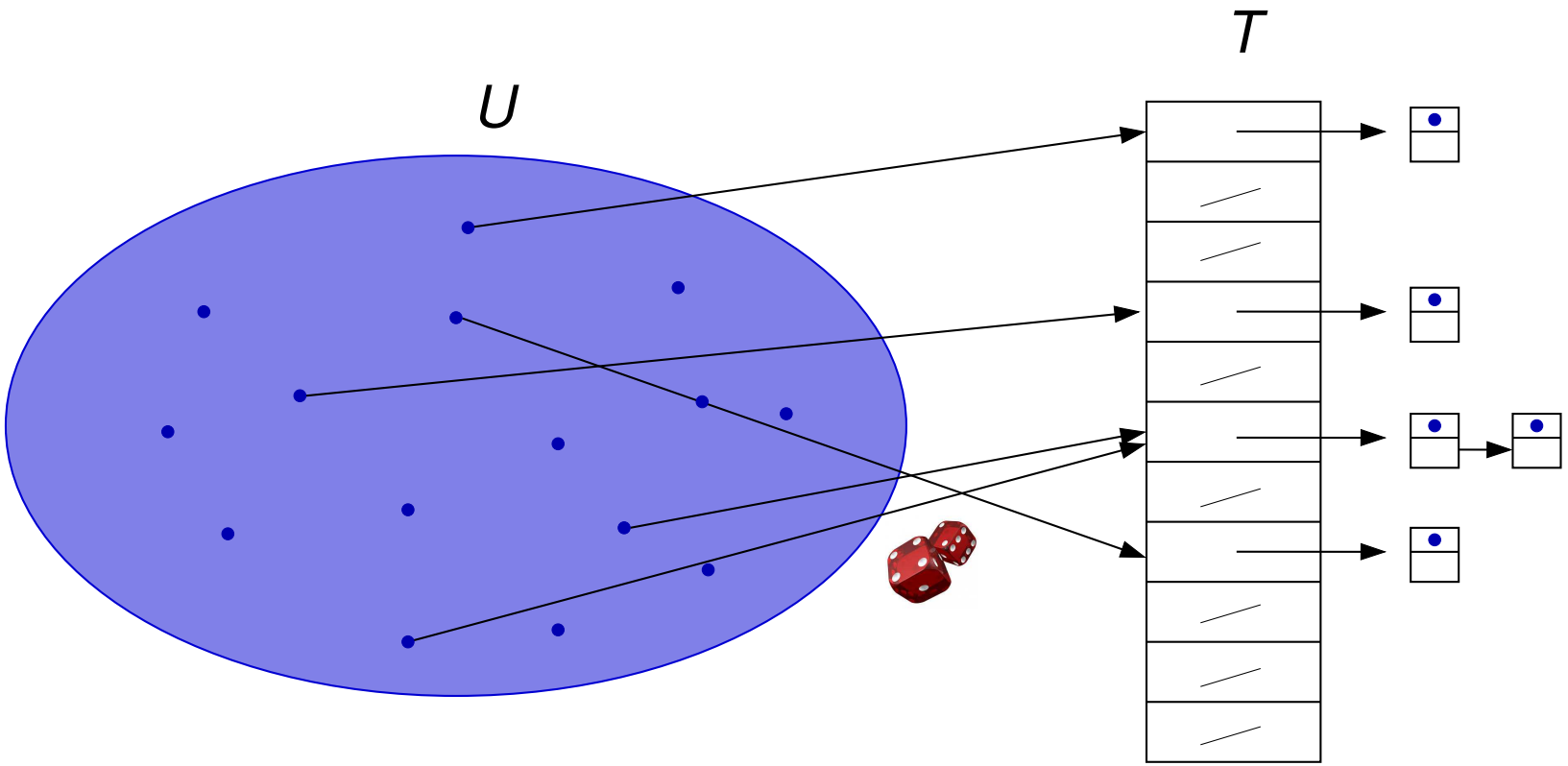

T

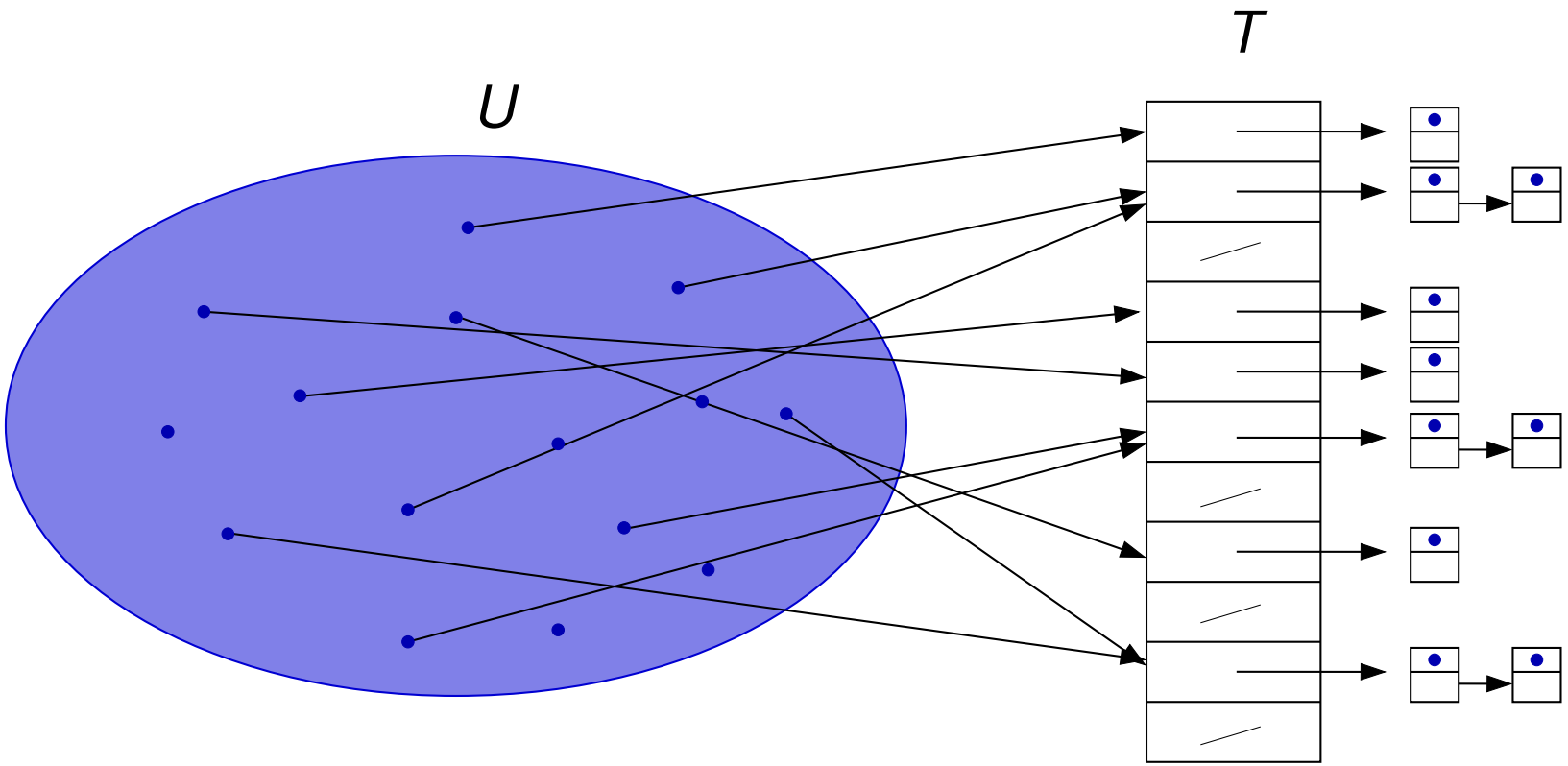

T
U


T
U


T
U


T
U

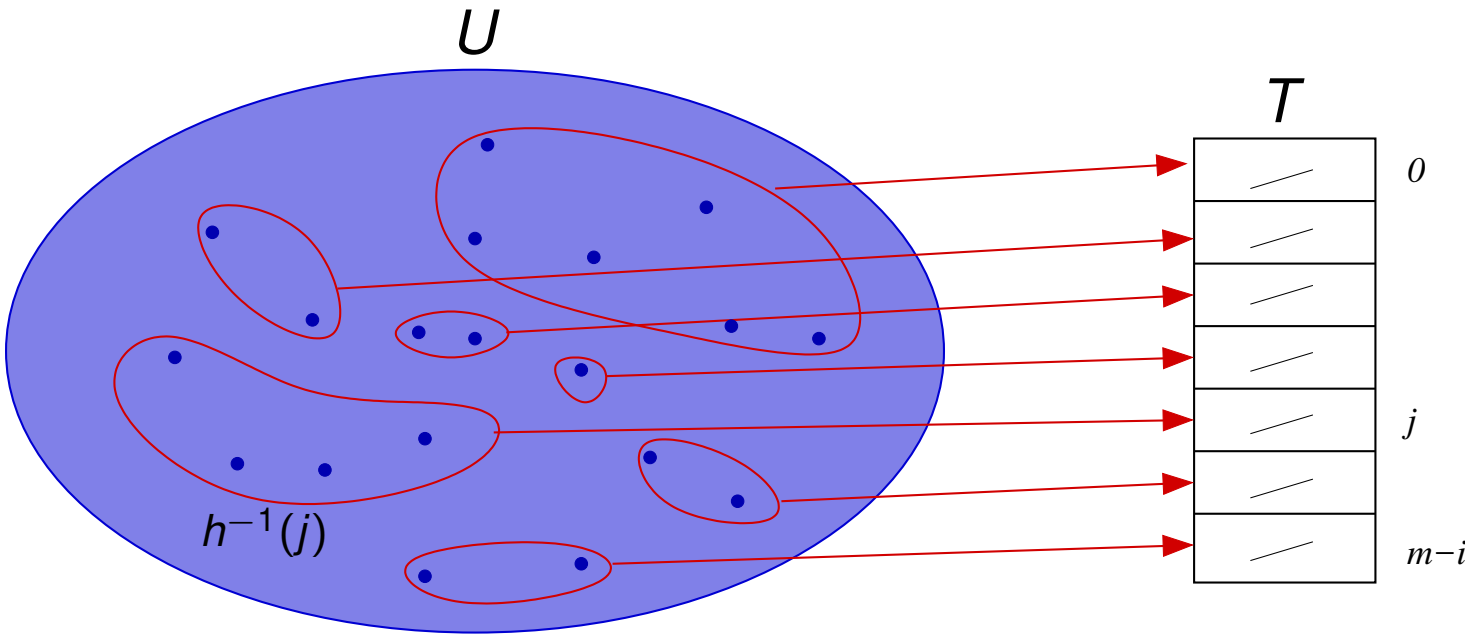

T
U



Simple Uniform Hashing: Keys are independently and with equal probability hashed into the m slots.

- this is a simplifying model for a hash function that “mixes keys and distributes them evenly”
- the actual hash function is still *deterministic*, not *randomized*

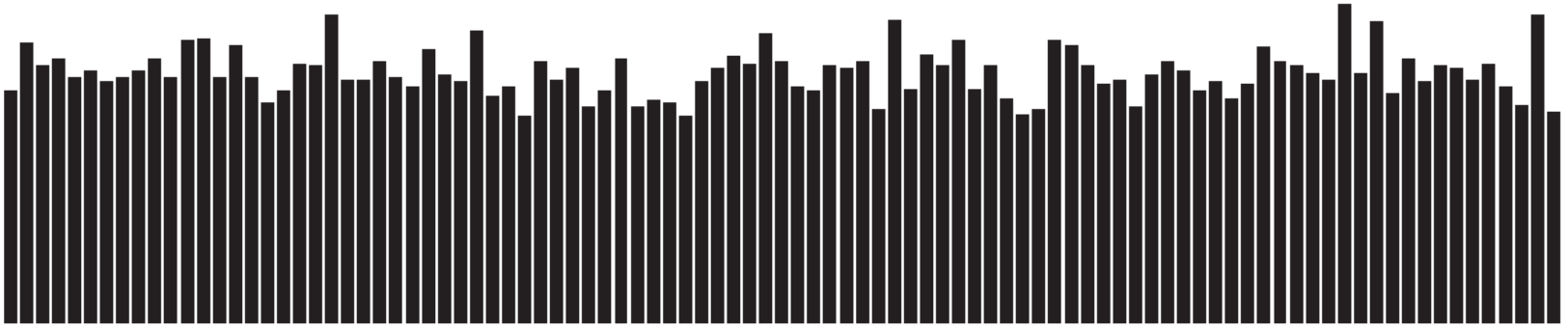
Alternative View:



Consider a fixed hash function h, and the sets

$$h^{-1}(j) = \{k \in U : h(k) = j\} \qquad (j = 0, \ldots, m-1)$$

Then: SUH $\sim$ keys selected for insertion are randomly drawn from U, with equal probability from each $h^{-1}(j)$.



Hash value frequencies for words in Tale of Two Cities (M = 97)

10679 different words appearing in Dickens's *Tale of Two Cities* hashed into 97 slots.

Source: R. Sedgewick and K. Wayne: Algorithms, 4th ed. Addison-Wesley 2011.

☞ diagram shows the combined effect of the *String.hashCode()* function, and the application of a hash function to the resulting key.

☞ distribution matches the expectations under the simple uniform hashing assumption.

Load Factor

Hashtable with $n = |K|$ elements in m slots has a *load factor*

$$\alpha = n/m$$

Average-case unsuccessful search

SUH assumption: search key hashes to a random slot of T.

Then: average (expected) search time is $\Theta(1 + \alpha)$.

Proof outline

A randomly chosen slot contains α keys on average (no assumption on distribution of keys over slots needed here). In unsuccessful search, all keys of the slot need to be compared to search key.

Average-case successful search

Assumption: search key is randomly selected from K, and distribution of keys in T follows SUH assumption.

Then: average (expected) search time is $\Theta(1 + \alpha)$.

Proof outline

Consider the key k_i that was added as the ith key to the hash table:

- $n - i$ keys were added after i
- of these, an average of $\frac{n-i}{m}$ are added in front of k_i in the linked list of k_i
- $\rightarrow$ expected time for searching for k_i: $1 + \frac{n-i}{m}$

$$\begin{aligned}
\text{Averaging over keys } k_i: \quad \frac{1}{n}\sum_{i=1}^{n}(1 + \frac{n-i}{m}) &= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{nm}(n^2 - \frac{(1+n)n}{2}) \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{\alpha}{2} + \frac{1}{2m}
\end{aligned}$$

Goal: the process

$$\textit{object } o \mapsto \ k = \textit{hashCode}(o) \mapsto \ h(k)$$

should satisfy the SUH assumption.

Leads to the general objective:

- avoid that "similar" objects are mapped to the same $h(k)$
- $\rightsquigarrow$ for h: make sure that $h(k)$ depends on all bits of k

Division Method

$$h(k) = k \bmod m$$

Choose m, so that

- n/m is an acceptable load factor
- m is a prime number not close to a power of 2.

Multiplication Method

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

defined by a constant $0 < A < 1$.

- Performs well for e.g. $A = (\sqrt{5} - 1)/2$
- Choice of m not critical, can choose powers of 2.

Universal Hashing

Simple uniform hashing:

- (conceptual) randomization over key selection, or slot assignment
- even under SUH assumption: for any h there exists a set of keys K, such that all $k \in K$ will be hashed to the same slot.

Example: Symbol Tables

Consider programmers Alice and Bob:

- Alice writes programs with identifiers

  a,b,c,. . . ,a2,b2,. . .

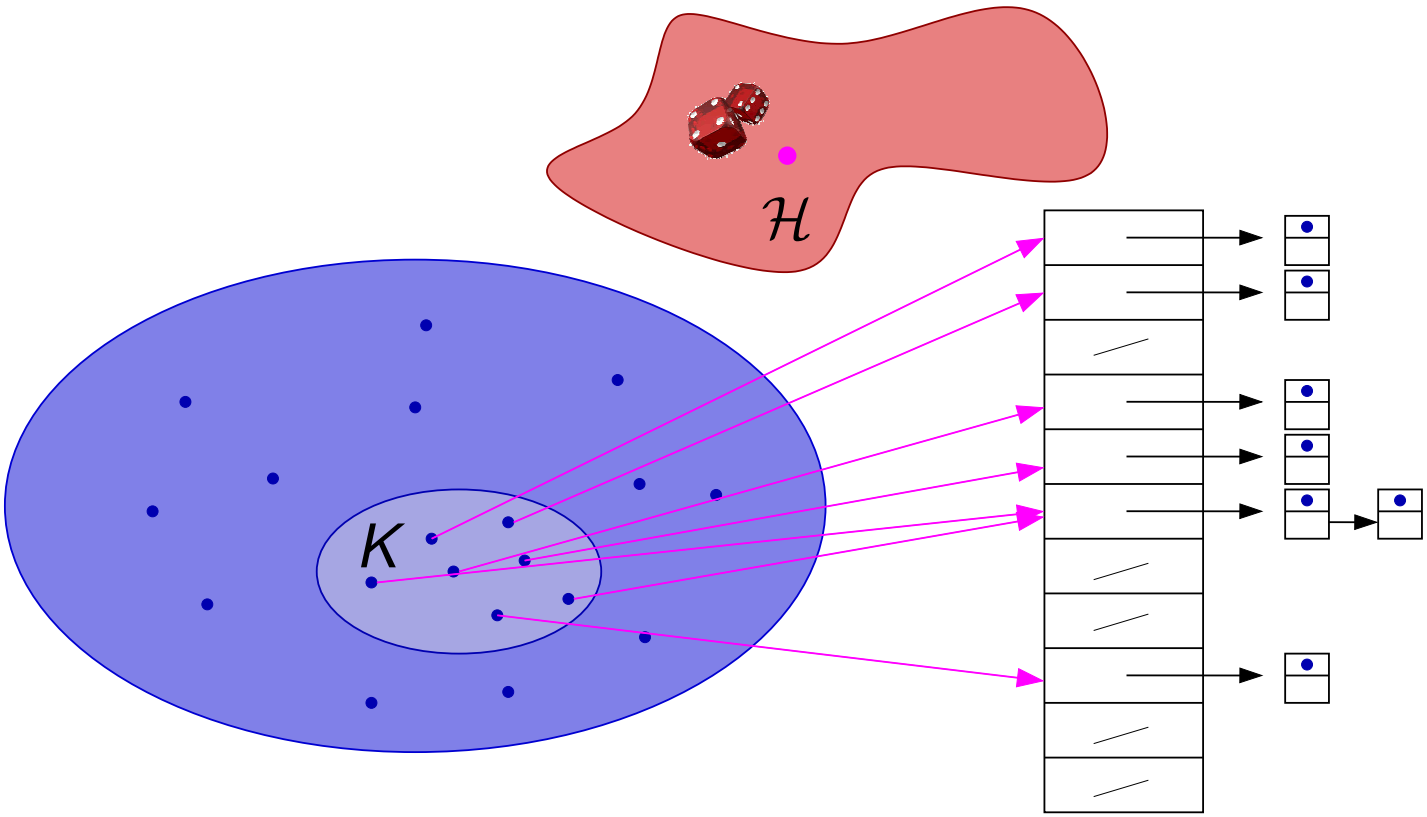
- Bob writes programs with identifiers

  integerIndexInWhileLoop, double4StoringAverage, . . .

Suppose the compiler constructs *symbol tables* using a certain hash function h. Then, if Alice's identifiers make a 'bad' set of keys for h, she will consistently observe a worse performance of the compiler than Bob.

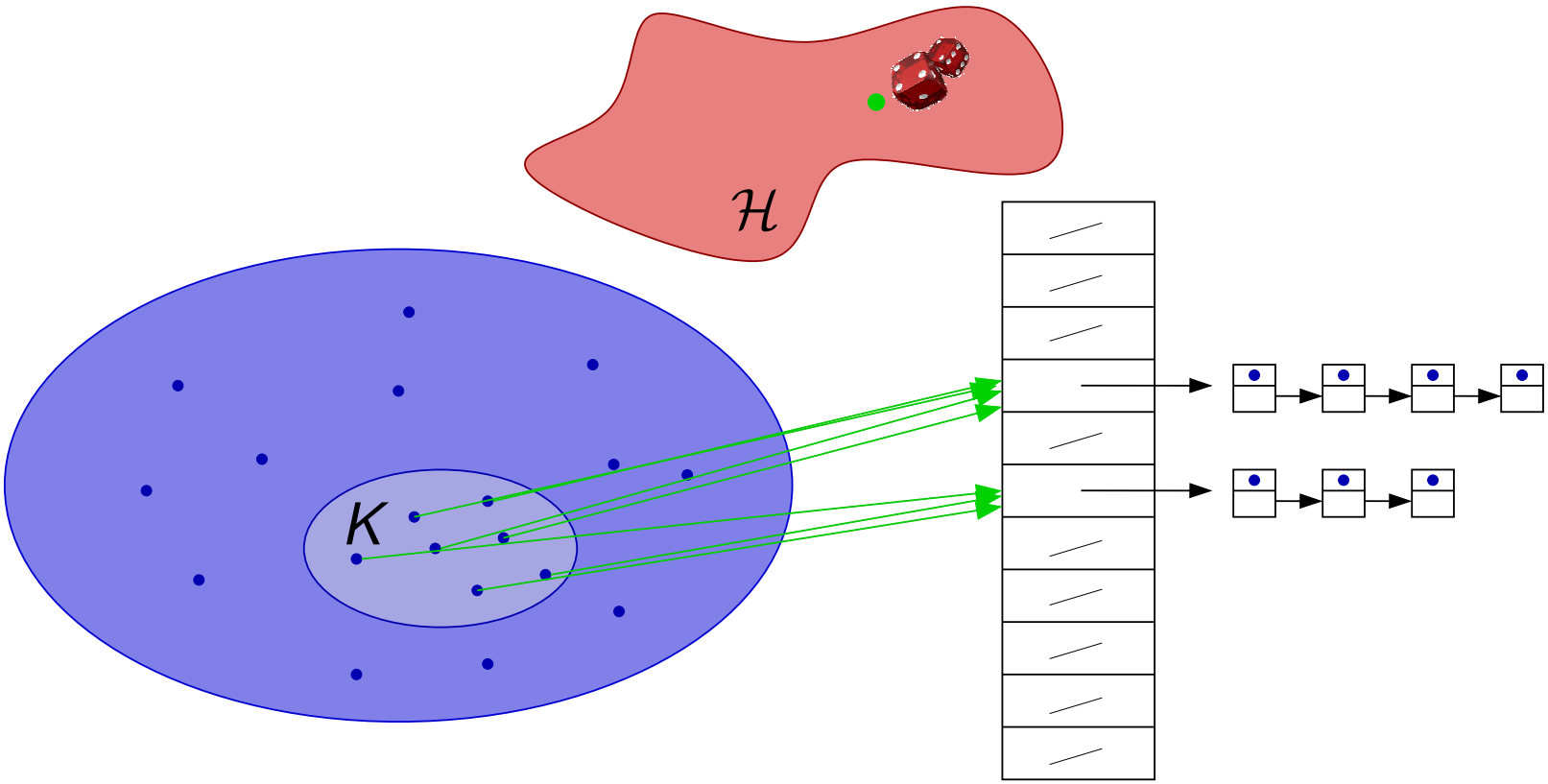
☞ try to average performance not over searches for a single h and T, but over multiple creations of hash tables.

Consider K fixed, but vary $h \in \mathcal{H}$:



Different $h \in \mathcal{H}$ lead to different distributions of the keys to the slots.

Consider K fixed, but vary $h \in \mathcal{H}$:



Different $h \in \mathcal{H}$ lead to different distributions of the keys to the slots.

Goal: for a fixed distribution over keys (or fixed set of keys) obtain a good average performance over many constructions of hash tables via random selection of h.

Universal Family

Family $\mathcal{H}$ of hash functions mapping U into $\{0, \ldots, m-1\}$ is called **universal**, if for any pair of keys $k, l \in U$:

$$|\{h \in \mathcal{H} : h(k) = h(l)\}| \leq \frac{|\mathcal{H}|}{m}$$

The Family $\mathcal{H}_{pm}$

- p: prime number larger than any $k \in U$
- $a \in \{1, \ldots, p-1\}$, $b \in \{0, \ldots, p-1\}$,

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$$

$$\mathcal{H}_{pm} = \{h_{ab} : 1 \leq a \leq p-1, 0 \leq b \leq p-1\}$$

Example

- $p = 13$, $U = \{0, 1, 2, \ldots 9\}$, $m = 3$

	key										$h_{ab}(k) \neq h_{ab}(l)$:		
a, b	0	1	2	3	4	5	6	7	8	9	$k = 0, l = 1$	$k = 4, l = 5$	$k = 4, l = 7$
$4, 6$	0	1	1	2	0	0	1	2	0	0	y	n	y
$1, 3$	0	1	2	0	1	2	0	1	2	0	y	y	n
$2, 8$	2	1	0	1	0	2	1	0	2	0	y	y	n
$1, 7$	1	2	0	1	2	0	0	1	2	0	y	y	y
$2, 10$	1	0	1	0	2	1	0	2	0	2	y	y	n

Theorem

$\mathcal{H}_{pm}$ is universal.

Theorem

- h randomly chosen from universal family of hash functions
- h used to hash n keys in hash table with chaining of size m

Then:

- for $k \notin T$: expected length of list at $T[h(k)]$ is at most α
- for $k \in T$: expected length of list at $T[h(k)]$ is at most $1 + \alpha$

Open Addressing

Principle: all objects are directly stored as table entries $T[i]$. When inserting key k, and target slot $T[h(k)]$ is already occupied, find an alternative available empty slot.

Probe Sequence

Generalize hash function as

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that for all $k \in U$:

$$\{h(k, 0), h(k, 1), \ldots, h(k, m-1)\} = \{0, 1, \ldots, m-1\}$$

Insert and Search

insert(k): find the first empty slot in the sequence $h(k, 0), h(k, 1), \ldots$ and insert k there. Throw *overflow* exception if all $h(k, i)$ $(0 \leq i \leq m-1)$ are occupied.

search(k): search sequence $h(k, 0), h(k, 1), \ldots$ until k found, or first empty slot is encountered.

Problem

When deleting a key, cannot just search for the key and re-set its slot to *null*: then subsequent *search* for other keys may incorrectly fail.

Solution

Mark slots from which keys were deleted with a special value *deleted*, and

- in *search*: treat *deleted* slots like occupied slots not containing the search key
- in *insert*: treat *deleted* slots like empty slots

☞ Complexity of *search* now depends on the total number of keys that were inserted at some time, not the actual number of keys currently in T.

Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Primary Clustering

Probability to be next occupied:

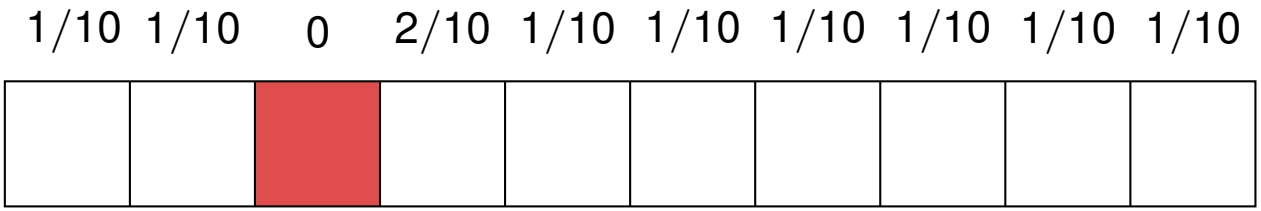
1/10	1/10	1/10	1/10	1/10	1/10	1/10	1/10	1/10	1/10

Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Primary Clustering

Probability to be next occupied:

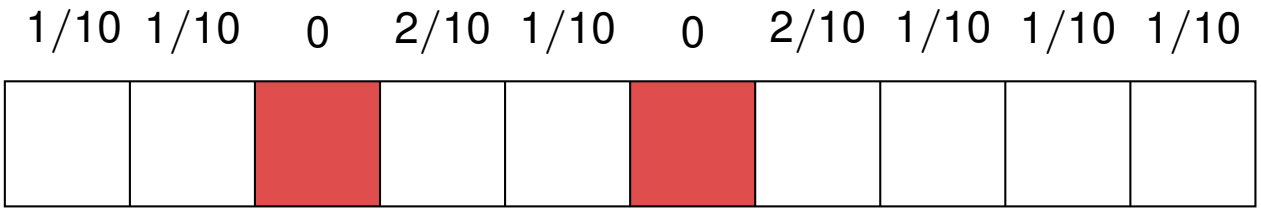


Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Primary Clustering

Probability to be next occupied:

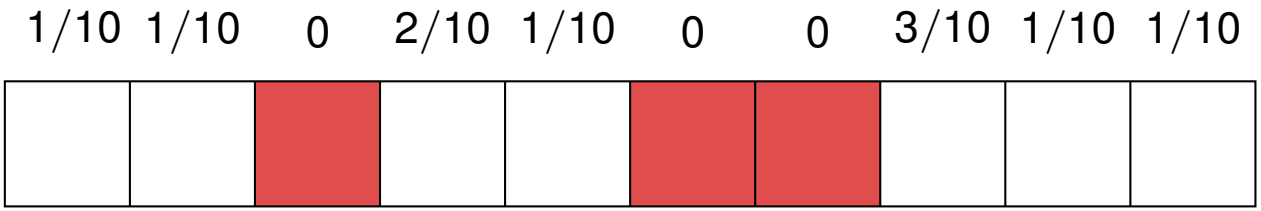


Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Primary Clustering

Probability to be next occupied:

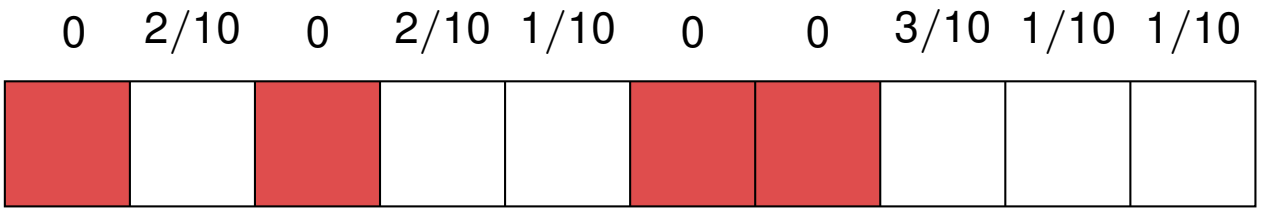


Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Primary Clustering

Probability to be next occupied:

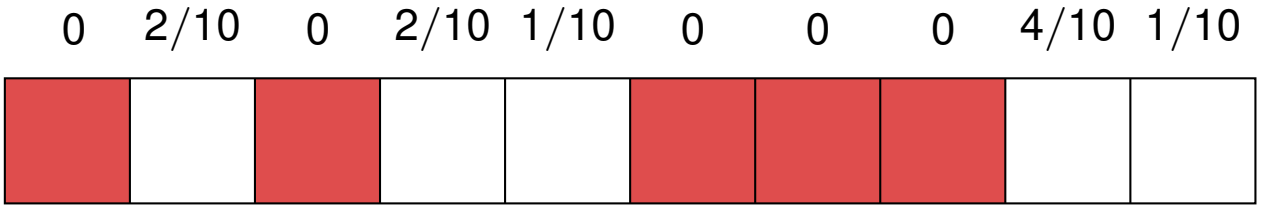


Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Primary Clustering

Probability to be next occupied:

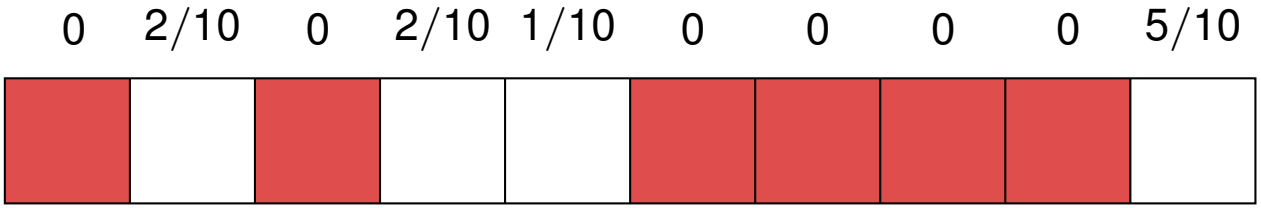


Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Primary Clustering

Probability to be next occupied:

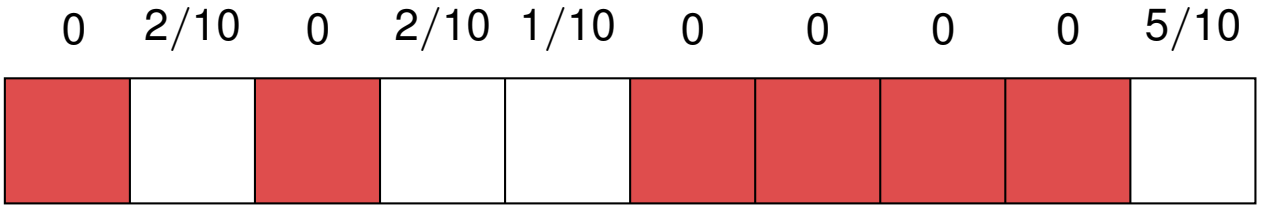


Based on standard hash function $h' : U \to \{0, \ldots, m-1\}$:

$$h(k, i) = (h'(k) + i) \bmod m$$

Primary Clustering

Probability to be next occupied:



Tendency to build contiguous runs of occupied slots – increases average *insert* and *search* times

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

(c_1, c_2, m must be chosen such that probe sequence is permutation of $\{0, 1, \ldots, m-1\}$.

☞ No primary clustering, but can still lead to buildup of "runs" of occupied slots when h' does not satisfy SUH assumption (**secondary clustering**).

Idea: to protect against secondary clustering, use a second auxiliary hash function for "diversification" of probe sequences.

Use two auxiliary hash functions h_1, h_2, and define:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

Ways to ensure that probe sequence is permutation of $\{0, 1, \ldots, m-1\}$:

- choose m as a power of 2, and h_2 so that it only produces odd numbers
- choose m as a prime number, and h_2 so that it only produces numbers $< m$.

Uniform Hashing

Assumption: every permutation of $\{0, 1, \ldots, m-1\}$ is equally likely to occur as the probe sequence.

The UH assumption *cannot* be satisfied by any of the hash functions considered so far:

- Linear, quadratic probing: at most m different probe sequences
- Double hashing: at most m^2 different probe sequences
- *but:* there exists $m!$ different permutations!

Analysis based on UH assumption still

- provides insight into the behavior of hash functions that “approximate” UH
- gives direction for constructing better hash functions

Theorem

Under the UH assumption, the expected number of probes in an unsuccessful search is $\leq \frac{1}{1-\alpha}$.

Proof outline

Imagine an infinite table with load factor α. Then, the probability that exactly k probes are required is

$$\alpha^{k-1}(1-\alpha)$$

and the expected number of probes is

$$\sum_{k=1}^{\infty} k\alpha^{k-1}(1-\alpha) = \sum_{k=1}^{\infty} \alpha^{k-1} = \frac{1}{1-\alpha}$$

Theorem

Under the UH assumption, and assuming that each key in T is equally likely to be searched for, the expected number of probes in a successful search is $\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.

Proof outline

Expected number of probes when searching for the key that was inserted as the ith element

$$=$$

Expected number of probes in an unsuccessful search in table with load factor $(i-1)/m$.

$$= \frac{1}{1-(i-1)/m}$$

Averaging over $i = 1, \ldots, n$ gives the bound.

α	Expected number of probes (upper bounds!) Unsuccessful	Successful
0.01	1.01	1.005
0.1	1.11	1.05
0.2	1.25	1.11
0.5	2.0	1.38
0.8	5.0	2.01
0.9	10	2.55
0.99	100	4.65