Algorithms

Dec 13, 2021

PROBABILISTIC ANALYSIS AND RANDOMIZED ALGORITHMS

Motivation

- Worst case does not usually happen
    - Average case analysis
    - Amortized analysis

Motivation

- Worst case does not usually happen
  - Average case analysis
  - Amortized analysis

- Randomization helps avoid worst-case and attacks by evil users
  - Choosing the pivot in quick-sort at random

Motivation

- Worst case does not usually happen
  - Average case analysis
  - Amortized analysis

- Randomization helps avoid worst-case and attacks by evil users
  - Choosing the pivot in quick-sort at random

Motivation

- Worst case does not usually happen
  - Average case analysis
  - Amortized analysis

- Randomization helps avoid worst-case and attacks by evil users
  - Choosing the pivot in quick-sort at random
- Randomization necessary in cryptography

Motivation

- Worst case does not usually happen
  - Average case analysis
  - Amortized analysis

- Randomization helps avoid worst-case and attacks by evil users
  - Choosing the pivot in quick-sort at random
- Randomization necessary in cryptography
- Can we get randomness?
  - How to extract randomness (extractors)
  - Longer "random behaving" strings from small seed (pseudorandom generators)

Probabilistic Analysis: The Hiring Problem

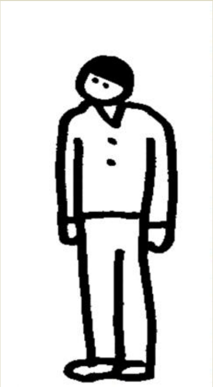
NY Knicks are going to hire one new basketball player

- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Example:



current best candidate

Probabilistic Analysis: The Hiring Problem

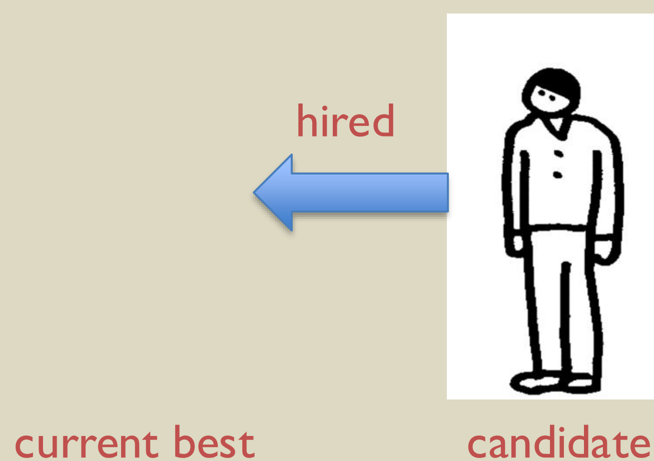
NY Knicks are going to hire one new basketball player

- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Example:



Probabilistic Analysis: The Hiring Problem

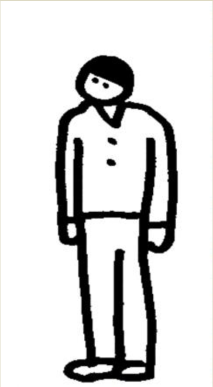
NY Knicks are going to hire one new basketball player

- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Example:



current best

candidate

Probabilistic Analysis: The Hiring Problem

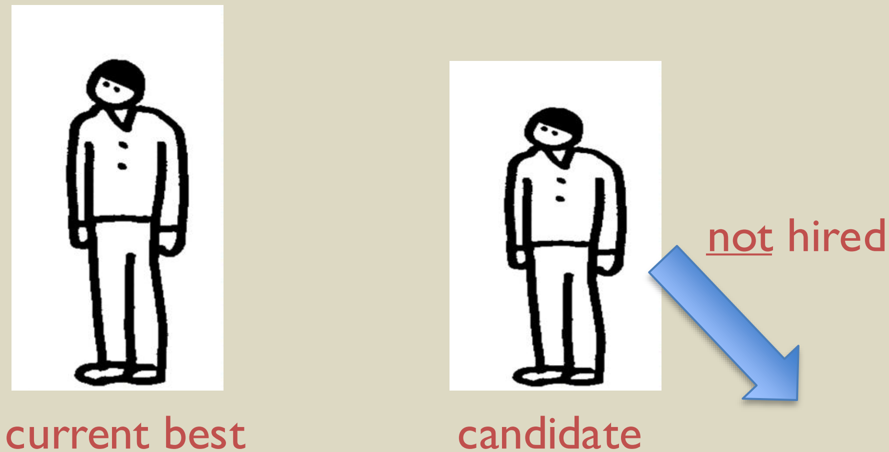
NY Knicks are going to hire one new basketball player

- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Example:



Probabilistic Analysis: The Hiring Problem

NY Knicks are going to hire one new basketball player

- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Example:

current best

candidate

Probabilistic Analysis: The Hiring Problem

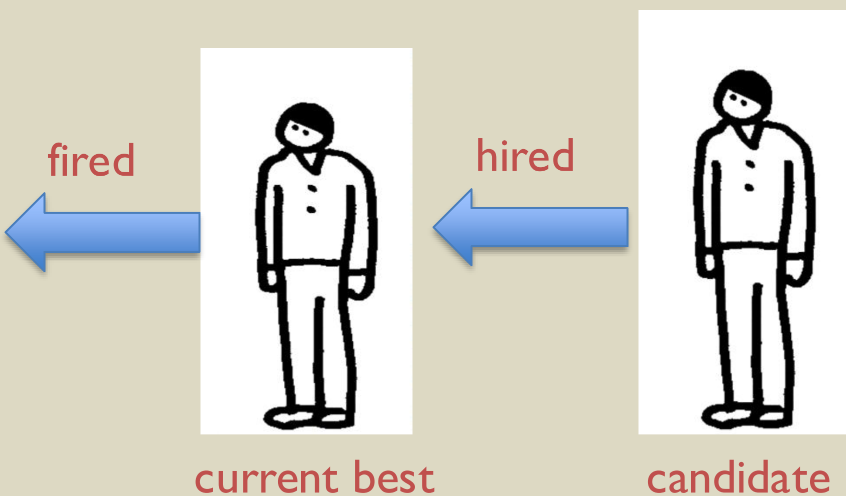
NY Knicks are going to hire one new basketball player

- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Example:



Probabilistic Analysis: The Hiring Problem

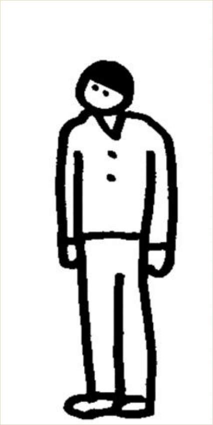
NY Knicks are going to hire one new basketball player

- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Example:



current best

candidate

Probabilistic Analysis: The Hiring Problem

NY Knicks are going to hire one new basketball player

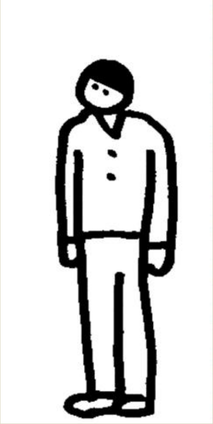
- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Question: **how many players did we (temporarily) hire?**

Example:



current best

candidate

Probabilistic Analysis: The Hiring Problem

NY Knicks are going to hire one new basketball player

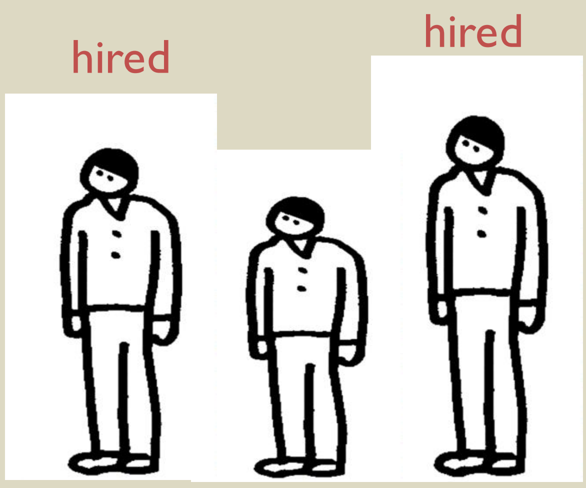
- the taller the better

They have **n** candidates that they call for interview

Strategy: each candidate is hired that is taller than the current best/tallest

Question: **how many players did we (temporarily) hire?**

Example:



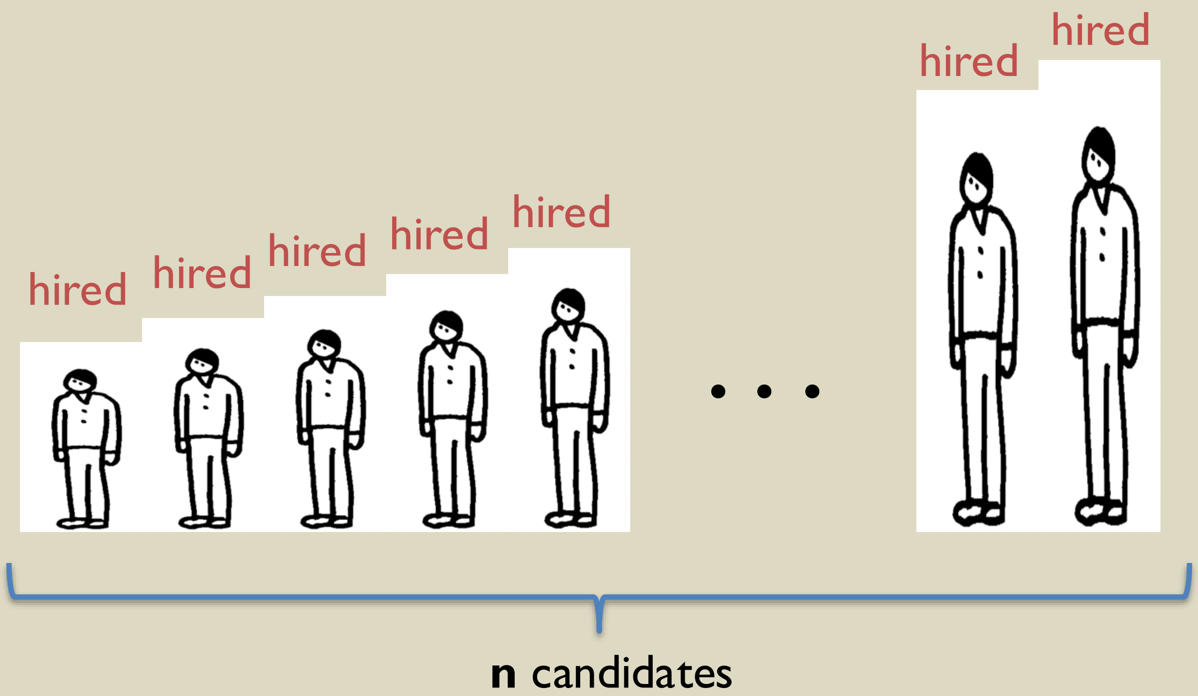
= 2

Worst-case analysis

In the **worst case**: how many players/candidates do we temporarily hire?

Worst-case analysis

In the **worst case**: how many players/candidates do we temporarily hire?



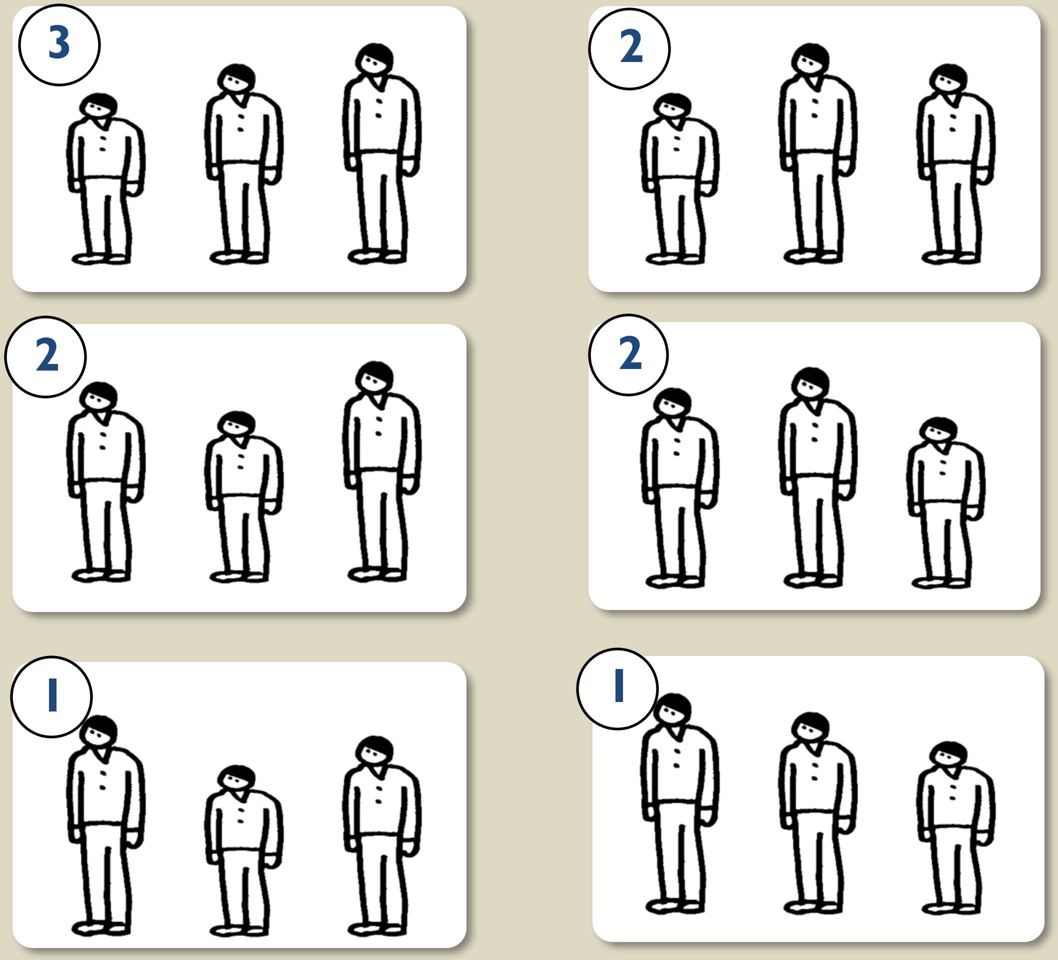
Answer: in the worst case we hire all **n** candidates

Worst-case unlikely to happen

- We only hire all candidates if they arrive in a specific order
- They are likely to arrive in a random order
- More interesting question (probabilistic analysis):

What is the expected number of hires we make over all the permutations of the candidates?

Example



Expected number of hires=

$$\frac{3+2+2+2+1+1}{6}$$

which equals 1+5/6

Calculating the expectation in general 1st trial

- **n!** permutations each equally likely
- Expectation = sum of hires in each permutation divided by **n!**

$$\frac{A_1 + A_2 + \cdots + A_{n!}}{n!}$$

Calculating the expectation in general 1st trial

- **n!** permutations each equally likely
- Expectation = sum of hires in each permutation divided by **n!**

$$\frac{A_1 + A_2 + \cdots + A_{n!}}{n!}$$

- For **n=5** we have **120** terms

Calculating the expectation in general 1st trial

- **n!** permutations each equally likely
- Expectation = sum of hires in each permutation divided by **n!**

$$\frac{A_1 + A_2 + \cdots + A_{n!}}{n!}$$

- For **n=5** we have **120** terms
- For **n=10** we have **3 628 800** terms

Calculating the expectation in general 1st trial

- **n!** permutations each equally likely
- Expectation = sum of hires in each permutation divided by **n!**

$$\frac{A_1 + A_2 + \cdots + A_{n!}}{n!}$$

- For **n=5** we have **120** terms
- For **n=10** we have **3 628 800** terms

NEED A MORE CLEVER METHOD

Indicator Random Variables

- Simple yet powerful technique for computing the expected value
- In particular, in situations in which there may be dependence

Indicator Random Variables

- Simple yet powerful technique for computing the expected value
- In particular, in situations in which there may be dependence

DEFINITION: Given a sample space and an event **A**, we define the **indicator random variable**

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs,} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

Indicator Random Variables

DEFINITION: Given a sample space and an event **A**, we define the **indicator random variable**

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs,} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

Indicator Random Variables

DEFINITION: Given a sample space and an event **A**, we define the **indicator random variable**

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs,} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

LEMMA: For an event **A**, let $\mathbf{X_A = I\{A\}}$. Then $\mathbf{E[X_A] = Pr[A]}$

Indicator Random Variables

DEFINITION: Given a sample space and an event **A**, we define the **indicator random variable**

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs,} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

LEMMA: For an event **A**, let $\mathbf{X_A} = \mathbf{I\{A\}}$. Then $\mathbf{E[X_A] = Pr[A]}$

PROOF: $\mathbf{E[X_A] = 1 * Pr\{A\} + 0 * Pr\{\overline{A}\} = Pr\{A\}}$

Simple Example: Coin Flip

Determine the expected number of heads when we flip a coin one time

Simple Example: Coin Flip

Determine the expected number of heads when we flip a coin one time

- Sample space is **{H,T}**
- **Pr{H} = Pr{T} = ½**

Simple Example: Coin Flip

Determine the expected number of heads when we flip a coin one time

- Sample space is **{H,T}**
- **Pr{H} = Pr{T} = ½**

- Define indicator variable $\mathbf{X_H = I\{H\}}$

Simple Example: Coin Flip

Determine the expected number of heads when we flip a coin one time

- Sample space is **{H,T}**
- **Pr{H} = Pr{T} = ½**

- Define indicator variable $\mathbf{X_H = I\{H\}}$
  - $\mathbf{X_H}$ counts the number of heads in one flip

Simple Example: Coin Flip

Determine the expected number of heads when we flip a coin one time

- Sample space is **{H,T}**
- **Pr{H} = Pr{T} = ½**

- Define indicator variable $\mathbf{X_H = I\{H\}}$
  - $\mathbf{X_H}$ counts the number of heads in one flip

- Since **Pr{H} = 1/2**, previous lemma says that $\mathbf{E[X_H] = 1/2}$

Slightly More Complex: **n** Coin Flips

Determine the expected number of heads when we flip n coins

- Let **X** be a random variable for the number of heads in **n** flips

Slightly More Complex: **n** Coin Flips

Determine the expected number of heads when we flip n coins

- Let **X** be a random variable for the number of heads in **n** flips
- Could calculate

$$E[X] = \sum_{k=0}^{n} k \cdot \Pr\{X = k\}$$

- … but cumbersome
- Instead use indicator variables

Slightly More Complex: **n** Coin Flips

Determine the expected number of heads when we flip n coins

- Let **X** be a random variable for the number of heads in **n** flips
- For **i = 1, …, n**, define $\mathbf{X_i}$ **= I{the i'th flip results in event H}**

Slightly More Complex: **n** Coin Flips

Determine the expected number of heads when we flip n coins

- Let **X** be a random variable for the number of heads in **n** flips

- For **i = 1, …, n**, define $\mathbf{X_i}$ **= I{the i'th flip results in event H}**
  - Then $\mathbf{E[X] = E[X_1 + X_2 + \ldots + X_n]}$

Slightly More Complex: **n** Coin Flips

Determine the expected number of heads when we flip n coins

- Let **X** be a random variable for the number of heads in **n** flips
- For **i = 1, …, n**, define $\mathbf{X_i}$ **= I{the i'th flip results in event H}**
  - Then $\mathbf{E[X] = E[X_1 + X_2 + \ldots + X_n]}$
- By linearity of expectation i.e., that $\mathbf{E[aX + bY] = aE[X] + bE[Y]}$

  *holds even if **X** and **Y** are dependent*

Slightly More Complex: **n** Coin Flips

Determine the expected number of heads when we flip n coins

- Let **X** be a random variable for the number of heads in **n** flips

- For $i = 1, \ldots, n$, define $X_i = I\{\text{the i'th flip results in event H}\}$
  - Then $E[X] = E[X_1 + X_2 + \ldots + X_n]$

- By linearity of expectation i.e., that $E[aX + bY] = aE[X] + bE[Y]$

  *holds even if **X** and **Y** are dependent*

$$E[X] = E[X_1 + X_2 + \ldots + X_n] = E[X_1] + E[X_2] + \ldots + E[X_n]$$

Slightly More Complex: **n** Coin Flips

Determine the expected number of heads when we flip n coins

- Let **X** be a random variable for the number of heads in **n** flips
- For **i = 1, …, n**, define $\mathbf{X_i}$ **= I{the i'th flip results in event H}**
  - Then $\mathbf{E[X] = E[X_1 + X_2 + \ldots + X_n]}$
- By linearity of expectation i.e., that

$$\mathbf{E[aX + bY] = aE[X] + bE[Y]}$$

*holds even if **X** and **Y** are dependent*

$$\mathbf{E[X] = E[X_1 + X_2 + \ldots + X_n] = \underbrace{E[X_1]} + E[X_2] + \ldots + E[X_n]}$$

By Lemma equals Pr{H} = 1/2

Slightly More Complex: **n** Coin Flips

Determine the expected number of heads when we flip n coins

- Let **X** be a random variable for the number of heads in **n** flips

- For **i = 1, …, n**, define $\mathbf{X_i}$ **= I{the i'th flip results in event H}**
  - Then $\mathbf{E[X] = E[X_1 + X_2 + \ldots + X_n]}$

- By linearity of expectation i.e., that $\mathbf{E[aX + bY] = aE[X] + bE[Y]}$

  *holds even if **X** and **Y** are dependent*

$$\mathbf{E[X] = E[X_1 + X_2 + \ldots + X_n] = \underbrace{E[X_1]} + E[X_2] + \ldots + E[X_n]}$$

By Lemma equals Pr{H} = 1/2

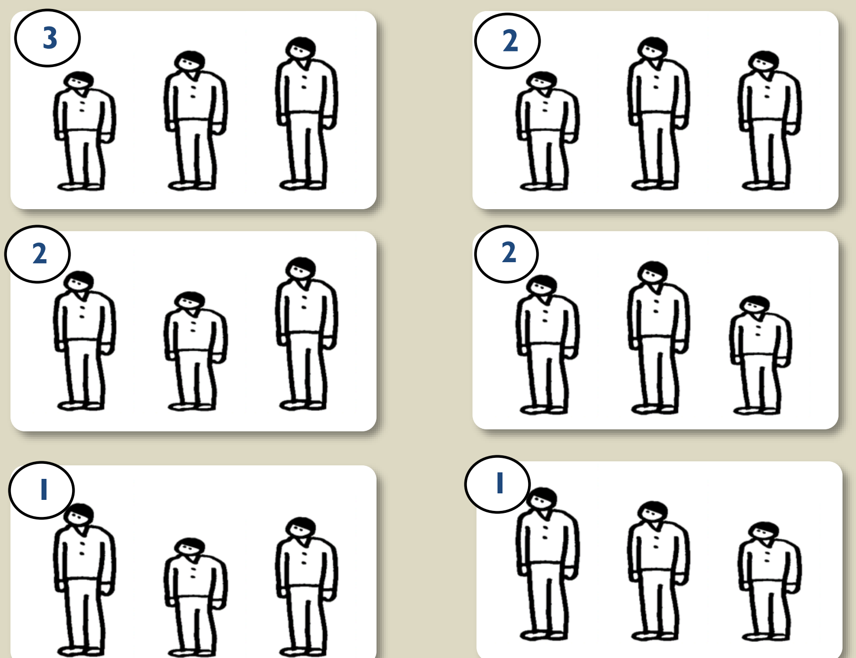
$$\mathbf{= n/2}$$

Probabilistic Analysis of Hiring Problem

- Candidates arrive in random order
- Let **X** be a random variable that equals the number of time we hire a player

Probabilistic Analysis of Hiring Problem

- Candidates arrive in random order
- Let **X** be a random variable that equals the number of times we hire a player



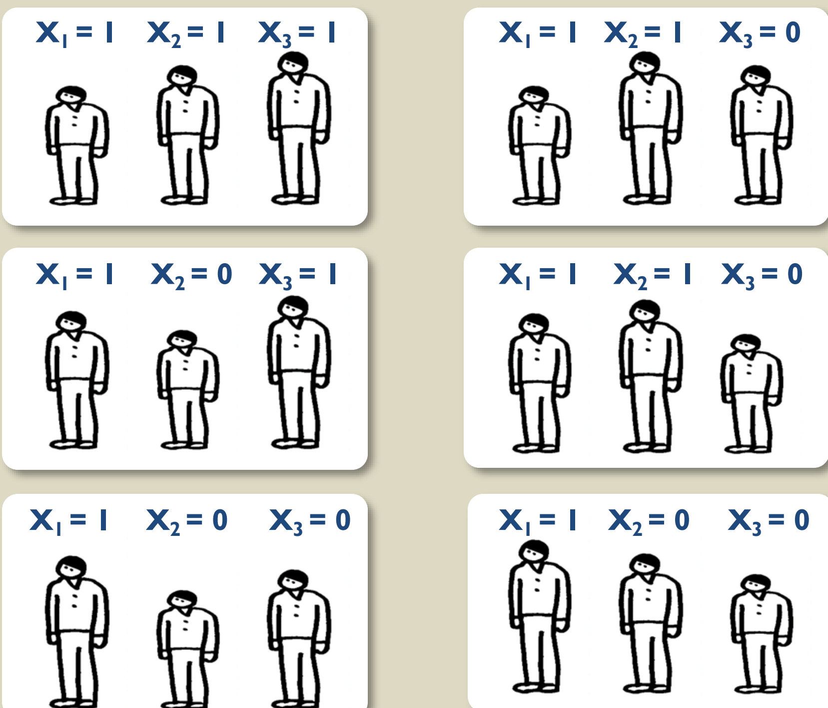
$$\mathbf{E[X]} = \frac{3+2+2+2+1+1}{6}$$

which equals 1+5/6

Probabilistic Analysis of Hiring Problem

- Candidates arrive in random order
- Let **X** be a random variable that equals the number of time we hire a player
- Define indicator variables $\mathbf{X_1, X_2, \ldots, X_n}$ where

$$\mathbf{X_i = I\{candidate\ i\ is\ hired\}}$$



$$\mathbf{E[X] = E[X_1+X_2+X_3]}$$

$$\mathbf{= E[X_1] + E[X_2] + E[X_3]}$$

$$\mathbf{= 1 + 1/2 + 1/3 = 1 + 5/6}$$

Probabilistic Analysis of Hiring Problem

- Candidates arrive in random order
- Let **X** be a random variable that equals the number of time we hire a player
- Define indicator variables $\mathbf{X_1, X_2, \ldots, X_n}$ where

  $\mathbf{X_i = I\{candidate\ i\ is\ hired\}}$

- Note that $\mathbf{X = X_1 + X_2 + \ldots + X_n}$ and $\mathbf{E[X_i] = Pr}\{$candidate $\mathbf{i}$ is hired$\}$

Probabilistic Analysis of Hiring Problem

- Candidates arrive in random order
- Let **X** be a random variable that equals the number of time we hire a player
- Define indicator variables $\mathbf{X_1, X_2, \ldots, X_n}$ where

  $\mathbf{X_i = I\{candidate\ i\ is\ hired\}}$

- Note that $\mathbf{X = X_1 + X_2 + \ldots + X_n}$ and $\mathbf{E[X_i] = Pr}\{$candidate **i** is hired$\}$

- By linearity of expectation,

$$\mathbf{E[X] = E[X_1 + X_2 + \ldots + X_n] = E[X_1] + E[X_2] + \ldots + E[X_n]}$$

Probabilistic Analysis of Hiring Problem

- Candidates arrive in random order
- Let **X** be a random variable that equals the number of time we hire a player
- Define indicator variables $\mathbf{X_1, X_2, \ldots, X_n}$ where

  $\mathbf{X_i = I\{candidate\ i\ is\ hired\}}$
- Note that $\mathbf{X = X_1 + X_2 + \ldots + X_n}$ and $\mathbf{E[X_i] = Pr\{}$candidate $\mathbf{i}$ is hired$\mathbf{\}}$

- By linearity of expectation,

$$\mathbf{E[X] = E[X_1 + X_2 + \ldots + X_n] = E[X_1] + E[X_2] + \ldots + E[X_n]}$$

which equals

$\mathbf{Pr\{}$candidate **1** is hired$\mathbf{\}}$ **+** $\mathbf{Pr\{}$candidate **2** is hired$\mathbf{\}}$ **+ … +** $\mathbf{Pr\{}$candidate **n** is hired$\mathbf{\}}$

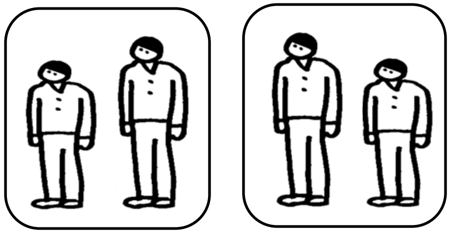
Probability of Hiring i'th Candidate

Pr{candidate **1** is hired**} = 1**

Probability of Hiring i'th Candidate

Pr{candidate **1** is hired**} = 1**

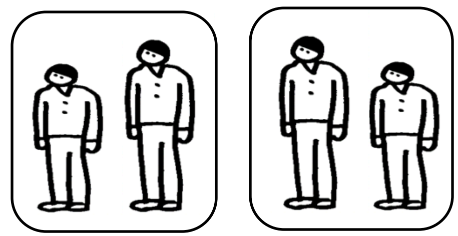
Pr{candidate **2** is hired**} = 1/2**



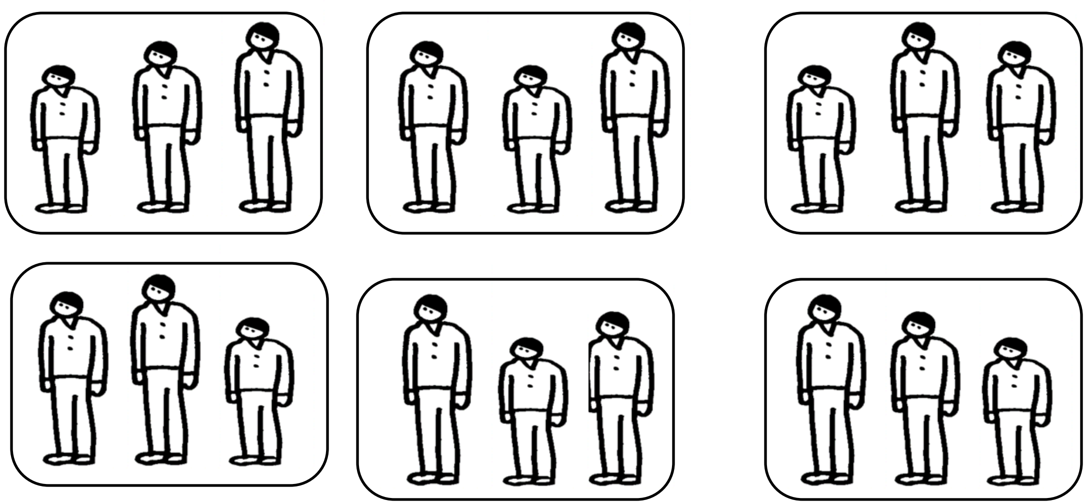
Probability of Hiring i’th Candidate

Pr{candidate **1** is hired} **= 1**

Pr{candidate **2** is hired} **= 1/2**



Pr{candidate **3** is hired} **= 1/3**

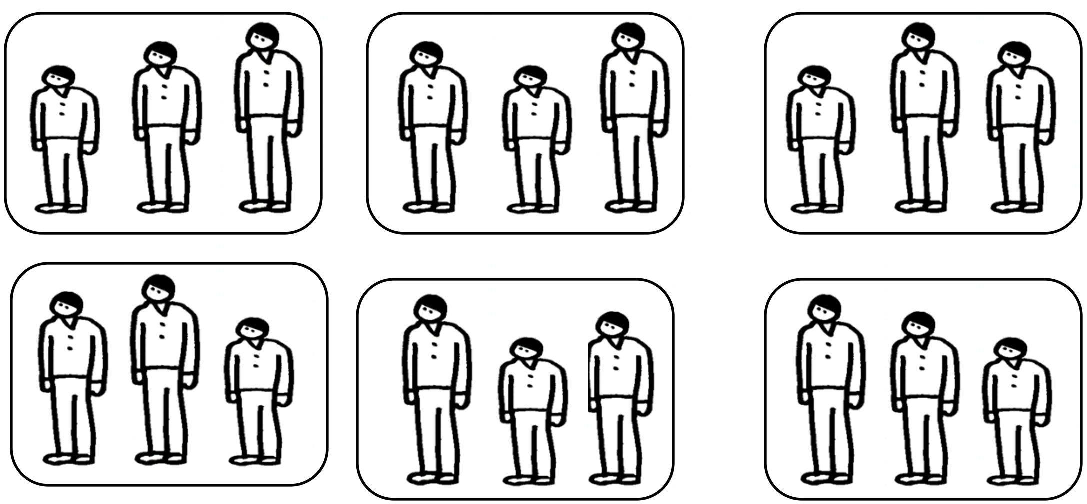


Probability of Hiring i’th Candidate

- i’th candidate hired iff he is tallest among the first i candidates
- Since they arrive in random order, any one of these first i candidates are equally likely to be the tallest =>

Pr{candidate **i** is hired**} = 1/i**

Pr{candidate **3** is hired**} = 1/3**



Expected Number of Hires

Recall that **E[number of hires] = E[X] =**

Pr{candidate **1** is hired**} + Pr{**candidate **2** is hired**} + … + Pr{**candidate **n** is hired**}**

Expected Number of Hires

Recall that **E[number of hires] = E[X] =**

Pr{candidate **1** is hired**} + Pr{**candidate **2** is hired**} + … + Pr{**candidate **n** is hired**}**

which equals

1/1 + 1/2 + 1/3 + 1/4 + 1/5 + … + 1/n

Expected Number of Hires

Recall that **E[number of hires] = E[X] =**

Pr{candidate **1** is hired**} + Pr{**candidate **2** is hired**} + … + Pr{**candidate **n** is hired**}**

which equals

$1/1 + 1/2 + 1/3 + 1/4 + 1/5 + \ldots + 1/n = H_n$

n:th harmonic number

Expected Number of Hires

Recall that **E[number of hires] = E[X] =**

Pr{candidate **1** is hired**} + Pr{**candidate **2** is hired**} + … + Pr{**candidate **n** is hired**}**

which equals

$$1/1 + 1/2 + 1/3 + 1/4 + 1/5 + \ldots + 1/n = H_n = \ln n + O(1)$$

n:th harmonic number

Expected Number of Hires

Recall that **E[number of hires] = E[X] =**

Pr{candidate **1** is hired**} + Pr{**candidate **2** is hired**} + … + Pr{**candidate **n** is hired**}**

which equals

$$1/1 + 1/2 + 1/3 + 1/4 + 1/5 + \ldots + 1/n = H_n = \ln n + O(1)$$

n:th harmonic number

Examples:

- Expected number of hires for **n=6** is **2.45**
- Expected number of hires for **n=100** is **5.1874**
- Expected number of hires for **n=10000** is **9.7876**

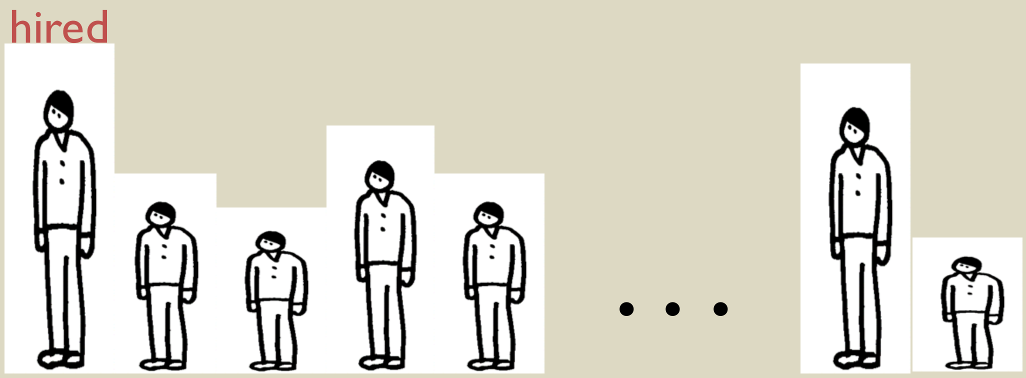
Questions

- What is the probability that we hire only one candidate?

- What is the probability that we hire **n** candidates?

Questions

- What is the probability that we hire only one candidate? **1/n** (tallest first)



- What is the probability that we hire **n** candidates? **1/n!** (worst case order)

Randomized Algorithm

- Instead of assuming that the candidates arrive in random order

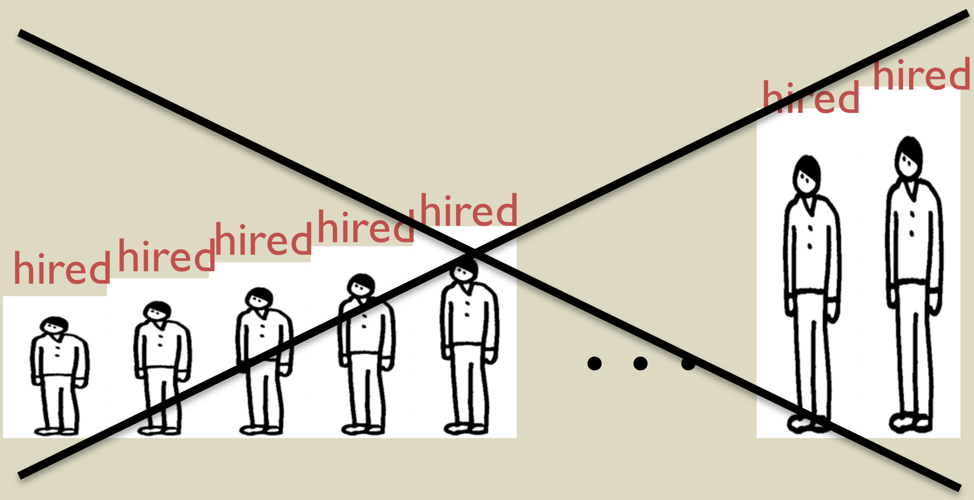
- **We/the algorithm** pick a random order and call the candidates in this order

Randomized Algorithm

- Instead of assuming that the candidates arrive in random order
- **We/the algorithm** pick a random order and call the candidates in this order

Randomized Algorithm

- Instead of assuming that the candidates arrive in random order
- **We/the algorithm** pick a random order and call the candidates in this order
- **In this way we can foul malicious users**



Question

- Given a function **RANDOM** that returns **1** with probability **p** and **0** with probability **1-p**

- How to use **RANDOM** for generating an unbiased bit?

Question

- Given a function **RANDOM** that returns **1** with probability **p** and **0** with probability **1-p**

- How to use **RANDOM** for generating an unbiased bit?

- Pick a pair (a,b) of random numbers: a = **RANDOM** and b = **RANDOM**
  - If a≠b return a
  - Otherwise pick a new pair

Two students in class have the same birthday with overwhelming probability

BIRTHDAY PARADOX

Birthday Paradox

- How many students in a room do we need so that the probability that two of them has the same birthday is at least 50%?

  (assuming each of the 365 days is equally likely)

Birthday Paradox

- How many students in a room do we need so that the probability that two of them has the same birthday is at least 50%?

  (assuming each of the 365 days is equally likely)

- Trivially: if we have 366 students then two of them has the same birthday with probability 1

Illustrative Example

2013 YEAR CALENDAR

JANUARY

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

FEBRUARY

S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28		

MARCH

S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

APRIL

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

MAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

JUNE

S	M	T	W	T	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

JULY

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

AUGUST

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

SEPTEMBER

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

OCTOBER

S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

NOVEMBER

S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

DECEMBER

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

Birthday Paradox

- How many students in a room do we need so that the probability that two of them has the same birthday is at least 50%?

  (assuming each of the 365 days is equally likely)

- Trivially: if we have 366 students then two of them has the same birthday with probability 1

- Surprisingly: 50% probability reached with 23 students and 99% probability reached with just 57 students

Birthday Lemma

If $q > 1.78\sqrt{|M|}$ then the probability that a function chosen uniformly at random $f: \{1, 2, \ldots, q\} \to M$ is injective is at most ½

Birthday Lemma

If $q > 1.78\sqrt{|M|}$ then the probability that a function chosen uniformly at random $f: \{1, 2, \ldots, q\} \rightarrow M$ is injective is at most ½

Proof.

Let $m = |M|$. The probability that the function is injective is

Birthday Lemma

If $q > 1.78\sqrt{|M|}$ then the probability that a function chosen uniformly at random $f: \{1, 2, \ldots, q\} \to M$ is injective is at most ½

Proof.

Let $m = |M|$. The probability that the function is injective is

Birthday Lemma

If **q > 1.78√|M|** then the probability that a function chosen uniformly at random **f: {1, 2, …, q} -> M** is injective is at most ½

Proof.

Let m = |M|. The probability that the function is injective is

$$\frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdots \frac{m-(q-1)}{m}$$

Since $e^{-x} > 1-x$ we have that this is less than

$$e^{-0} \cdot e^{-1/m} \cdot e^{-2/m} \cdots e^{-(q-1)/m} = e^{-q(q-1)/(2m)}$$

Birthday Lemma

If **q > 1.78√|M|** then the probability that a function chosen uniformly at random **f: {1, 2, …, q} -> M** is injective is at most ½

Proof.

Let m = |M|. The probability that the function is injective is

$$\frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdots \frac{m-(q-1)}{m}$$

Since $e^{-x} > 1-x$ we have that this is less than

$$e^{-0} \cdot e^{-1/m} \cdot e^{-2/m} \cdots e^{-(q-1)/m} = e^{-q(q-1)/(2m)}$$

which is less than ½ if $q \geq \frac{1+\sqrt{1+8\ln 2}}{2}\sqrt{m} \approx 1.78\sqrt{m}$

Birthday Lemma

If **$q > 1.78\sqrt{|M|}$** then the probability that a function chosen uniformly at random **f: {1, 2, …, q} -> M** is injective is at most ½

Proof.

Let m = |M|. The probability that the function is injective is

$$\frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdots \frac{m-(q-1)}{m}$$

Since $e^{-x} > 1-x$ we have that this is less than

$$e^{-0} \cdot e^{-1/m} \cdot e^{-2/m} \cdots e^{-(q-1)/m} = e^{-q(q-1)/(2m)}$$

which is less than ½ if $q \geq \frac{1+\sqrt{1+8\ln 2}}{2}\sqrt{m} \approx 1.78\sqrt{m}$

HASH FUNCTIONS AND TABLES

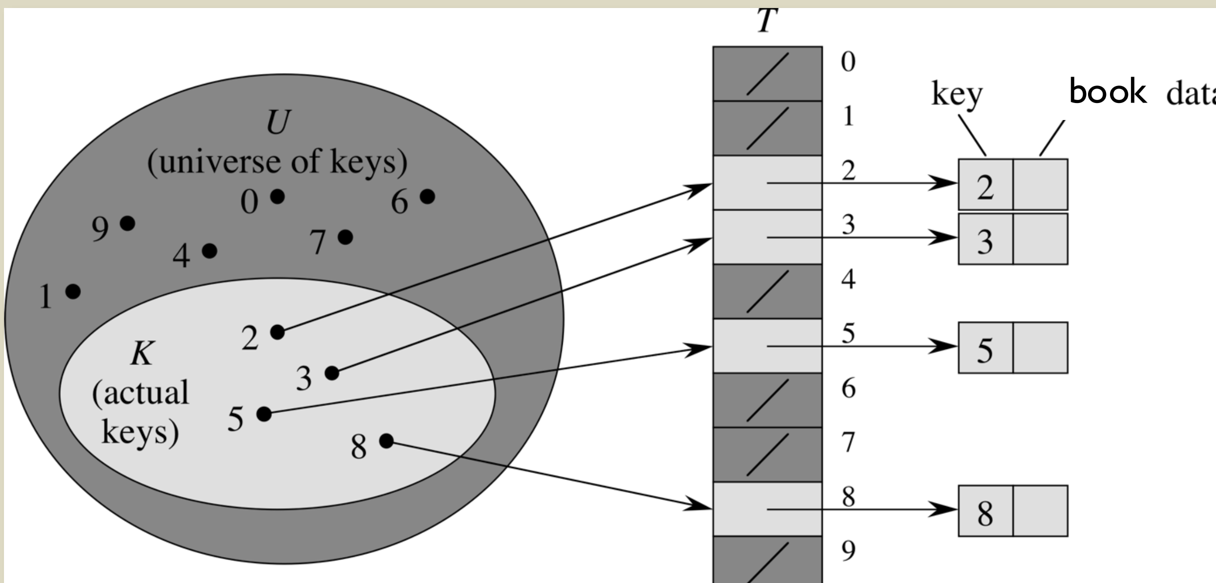
Design a Computer System for a Library

Design a Computer System for a Library

- Insert a new book
- Delete book
- Search book
- All operations in (expected) constant time!

Direct-Address Tables

- Simple technique that allows for simple implementation of constant-time insertion, deletion, and search
- Every book has one unique number (ISBN)
- Construct an array/table T with a position for each book



Direct-Address Tables

Direct-Address-Search(T,k):

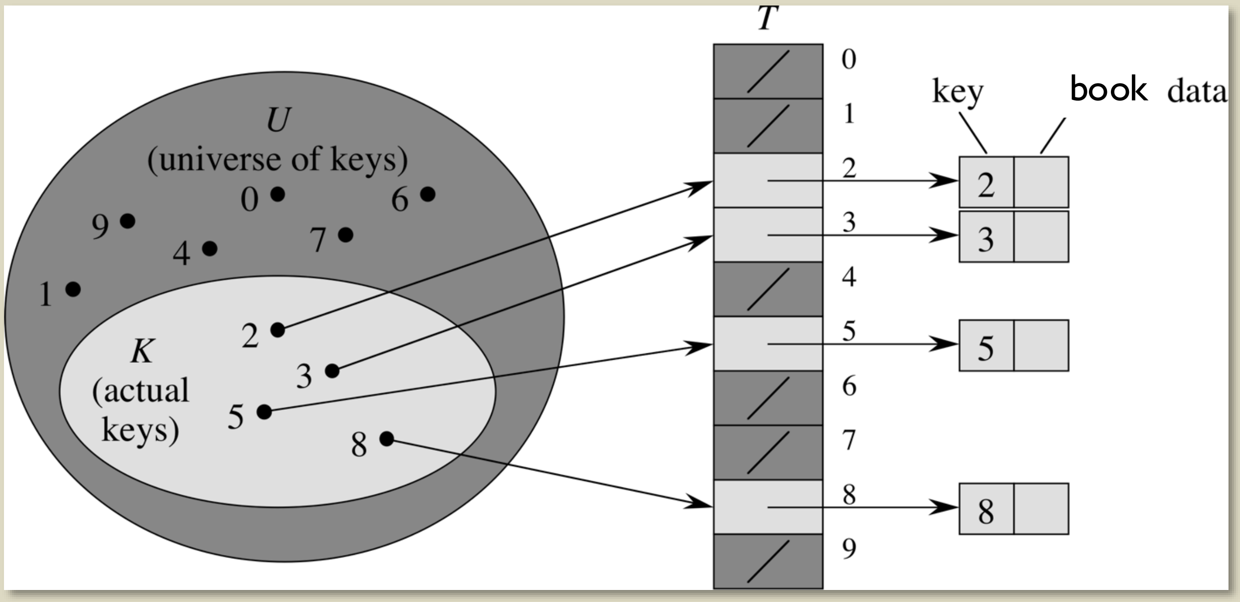
return T[k]

Direct-Address-Insert(T,x):

T[x.key] = x

Direct-Address-Delete(T,x):

T[x.key]= NIL



Direct-Address Tables

- Running time of each operation: **O(1)**

- Space: **O(|U|)**

Direct-Address-Search(T,k):

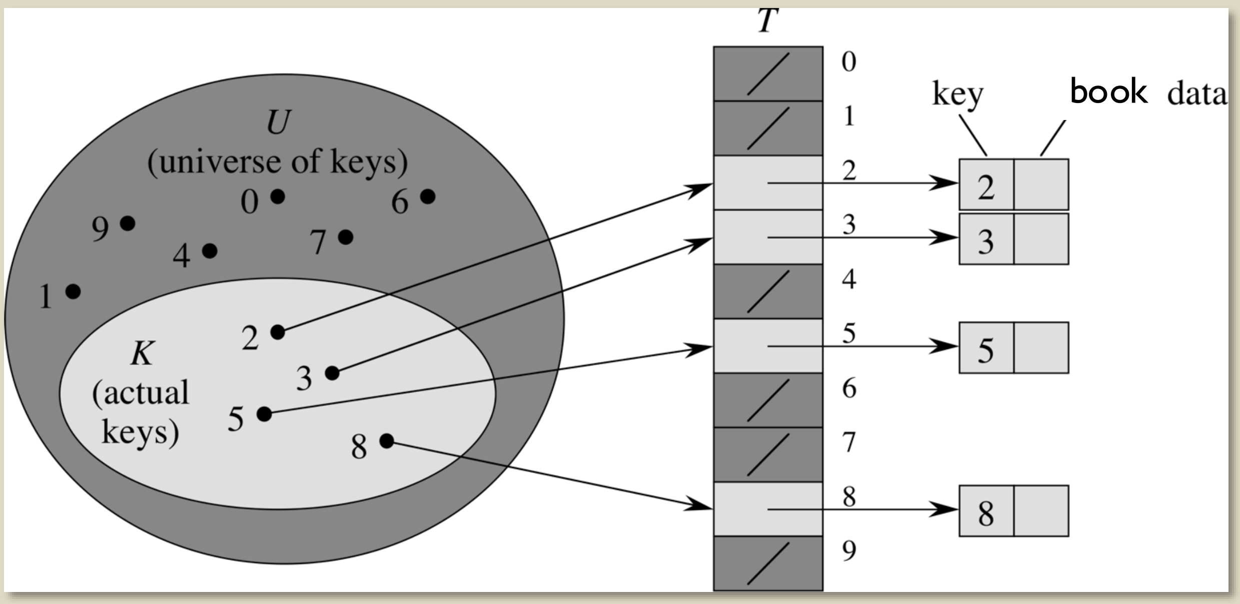
return T[k]

Direct-Address-Insert(T,x):

T[x.key] = x

Direct-Address-Delete(T,x):

T[x.key]= NIL



Direct-Address Tables

Positives

Negatives

Direct-Address Tables

Positives

- Running time of each operation: **O(1)**
- Easy implementation

Negatives

- Space: **O(|U|)**
- For most applications (like a Library) we only store a small fraction of all possible items
- Wish to use space proportional to the amount of information stored

Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case

Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **h(k)**

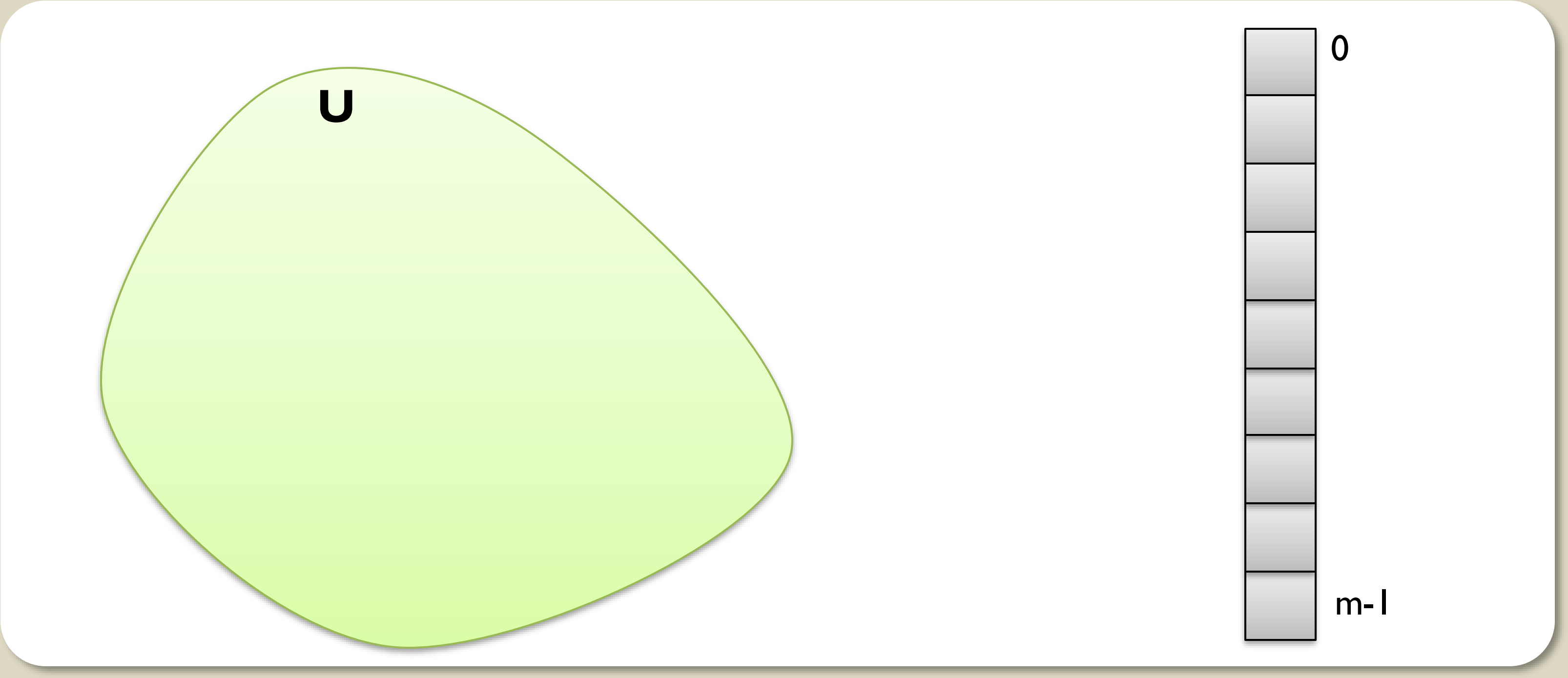
Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **$\Theta(K)$**
- Implement search, insertion, deletion in time **$O(1)$** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **$h(k)$**
- **$h: U \rightarrow \{0,1, \ldots, m-1\}$** is called the ***hash function***

table size

Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **h(k)**
- **h: U -> {0,1, …, m-1}** is called the ***hash function***

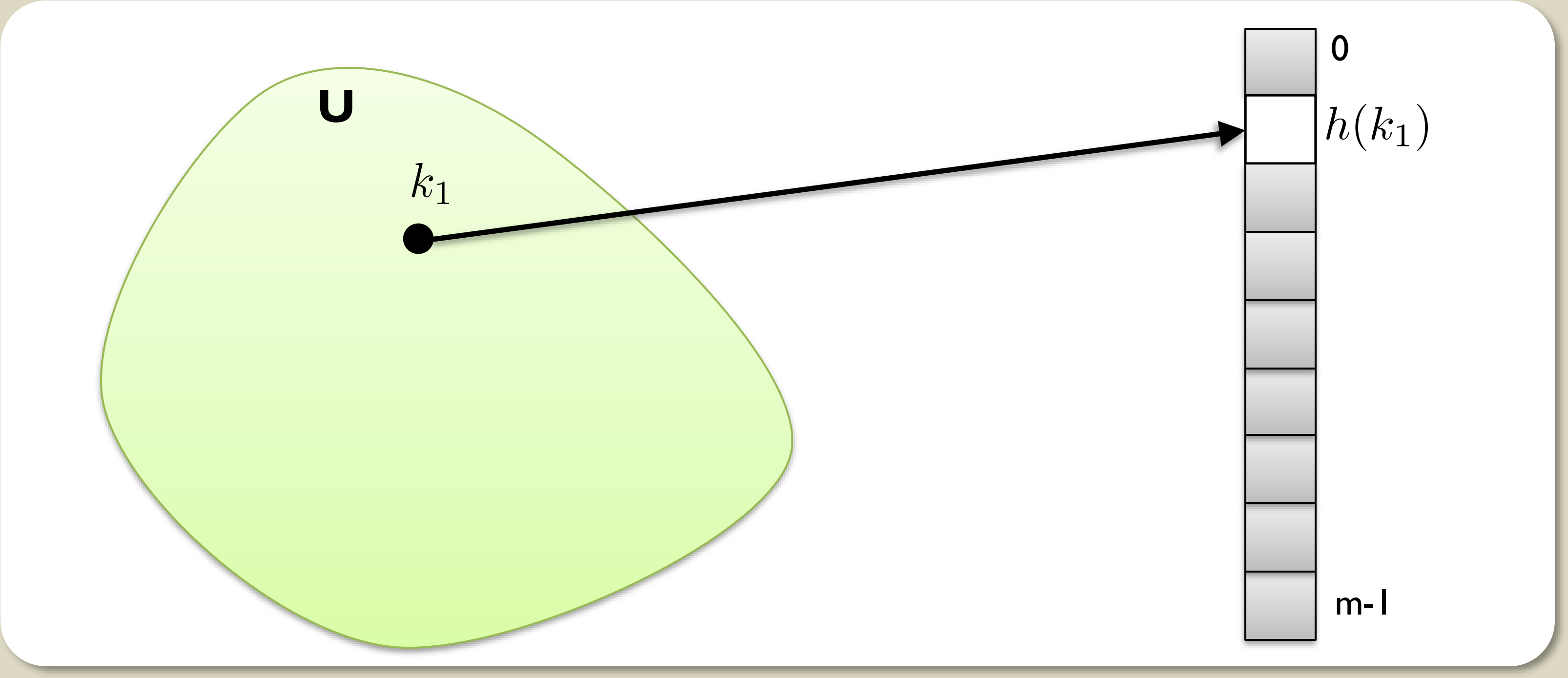


Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **h(k)**
- **h: U -> {0,1, …, m-1}** is called the ***hash function***

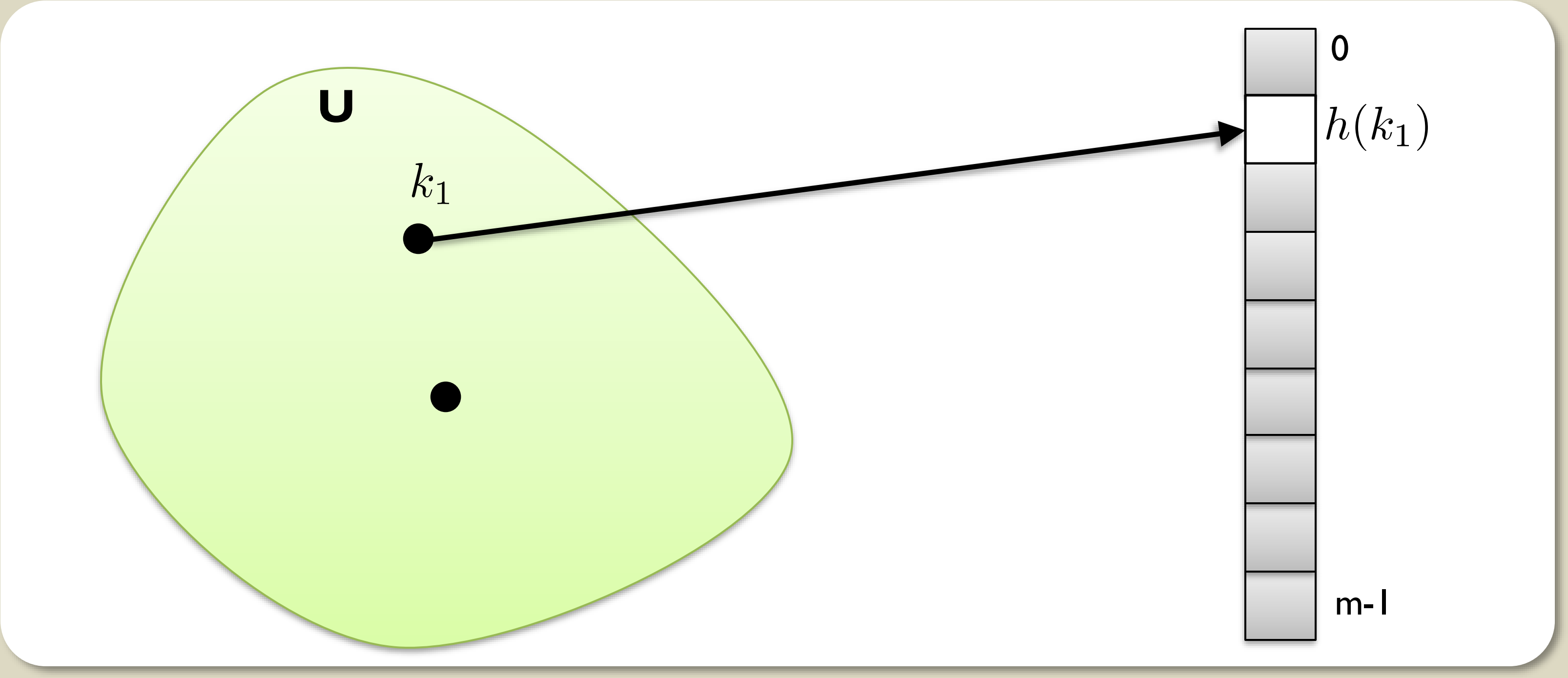
Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **h(k)**
- **h: U -> {0,1, …, m-1}** is called the ***hash function***



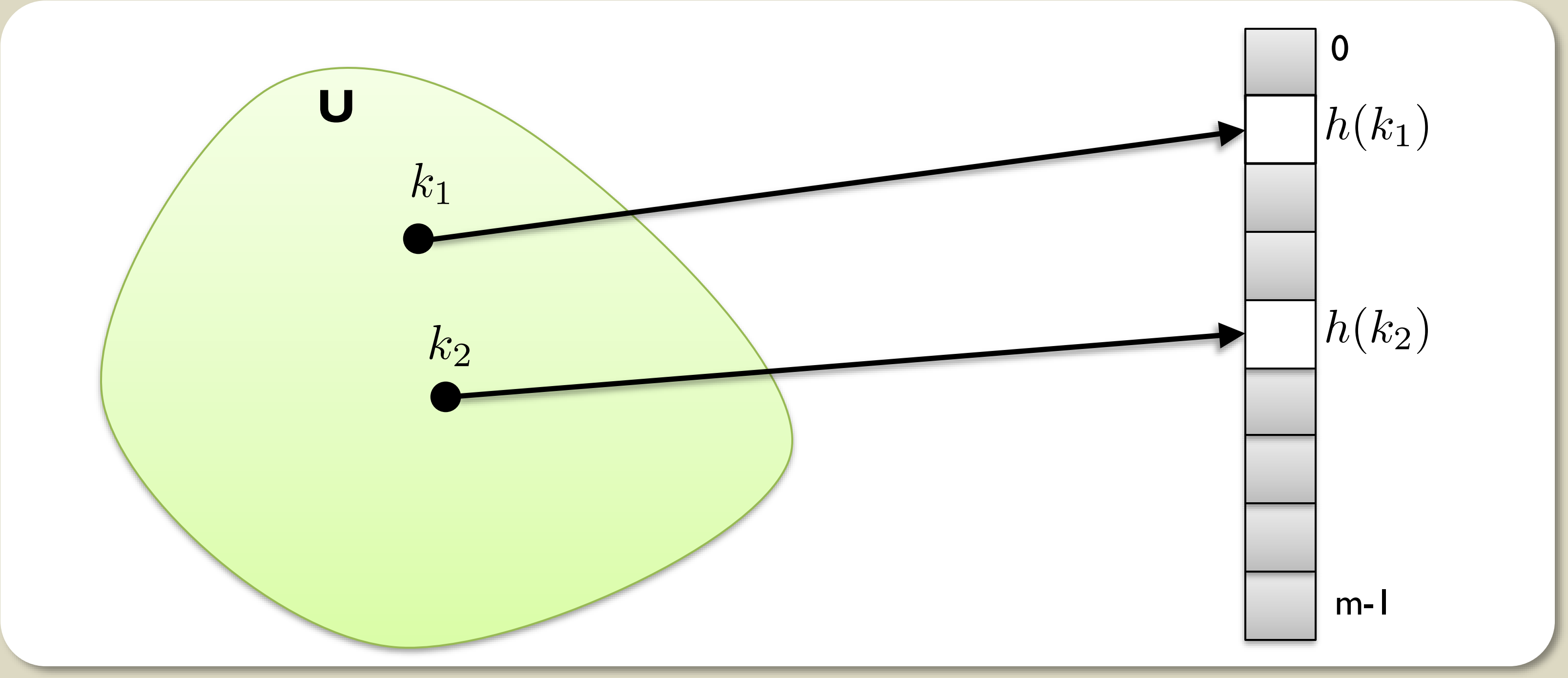
Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **h(k)**
- **h: U -> {0,1, …, m-1}** is called the ***hash function***



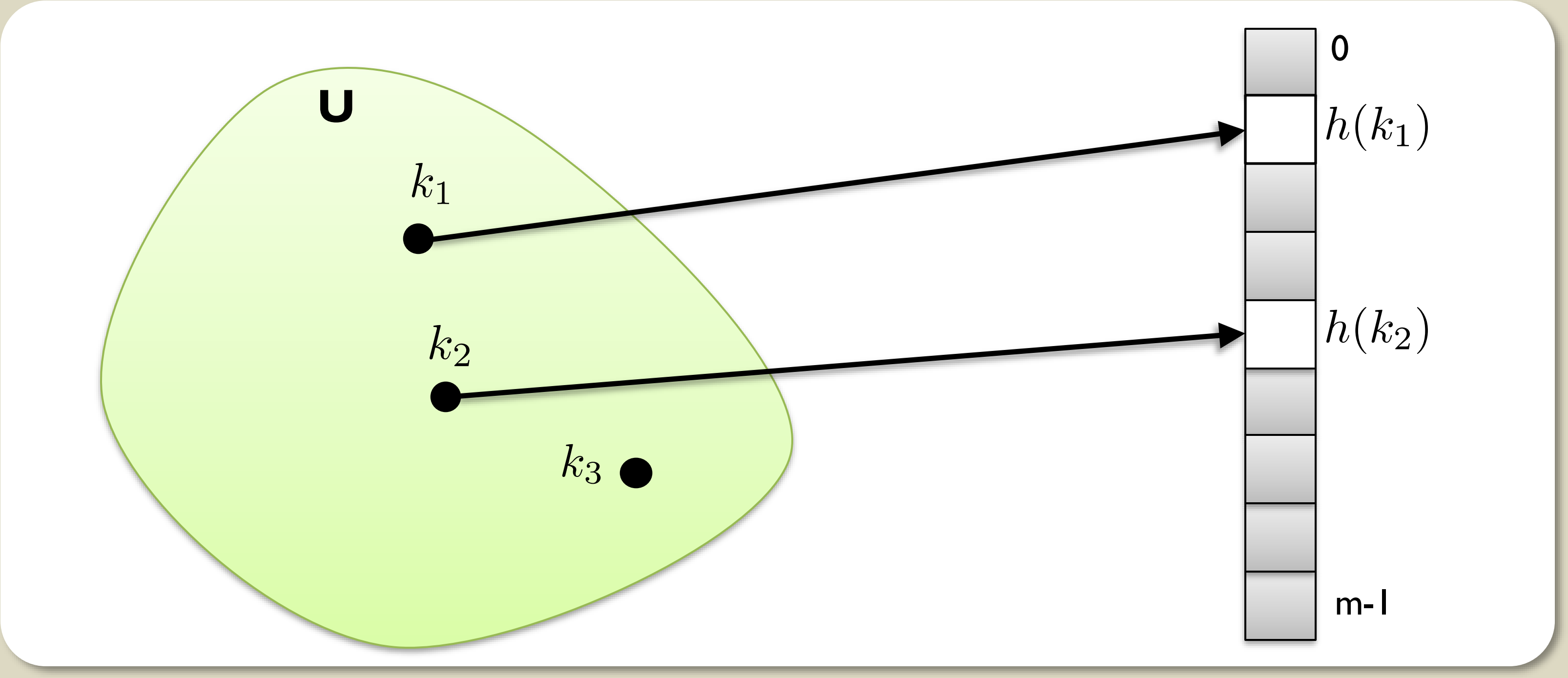
Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **h(k)**
- **h: U -> {0,1, …, m-1}** is called the ***hash function***



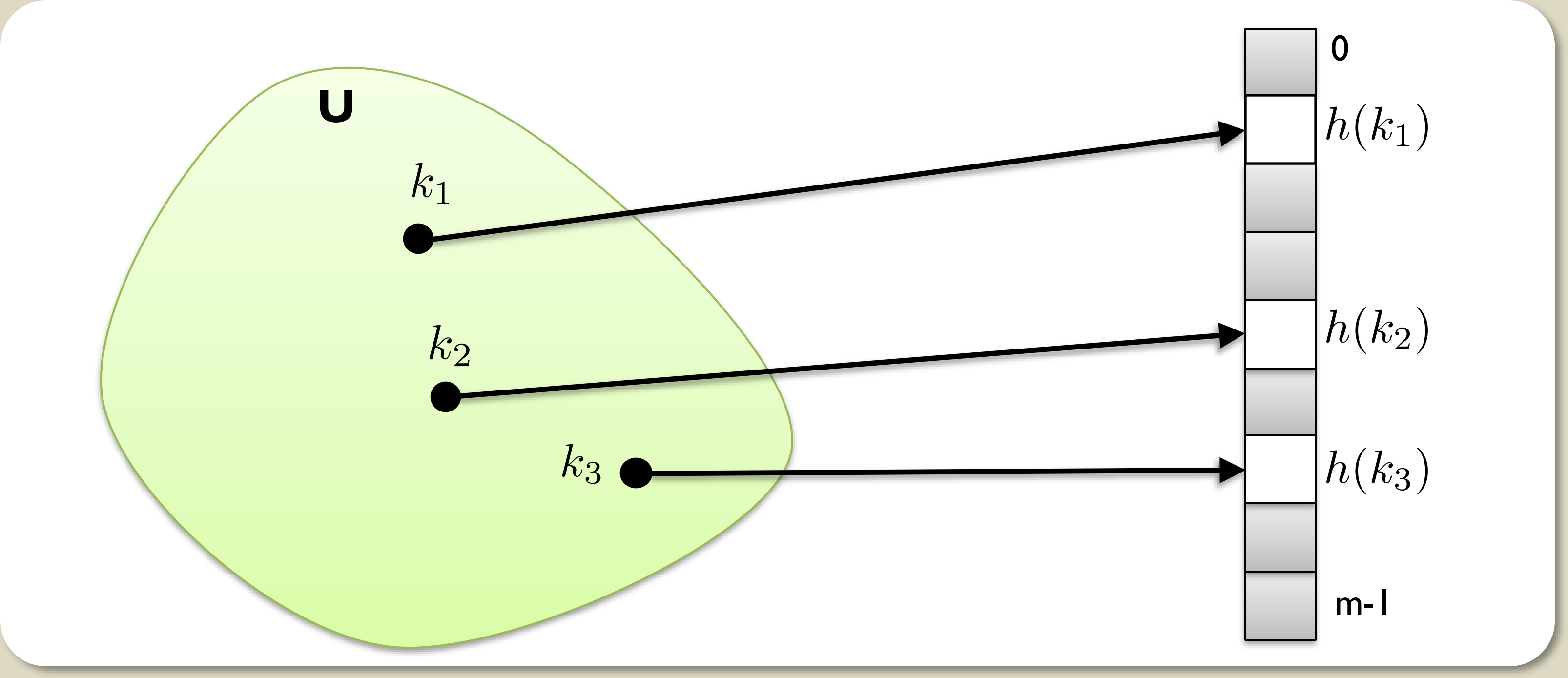
Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **h(k)**
- **h: U -> {0,1, …, m-1}** is called the ***hash function***



Hash Tables

- Uses space proportional to the number **K** of keys stored, i.e., **Θ(K)**
- Implement search, insertion, deletion in time **O(1)** in the average case
- In direct-address table an element with key **k** was stored in slot **k**
- In hash tables an element with key **k** is stored in slot **h(k)**
- **h: U -> {0,1, …, m-1}** is called the ***hash function***



Desired Properties of a Hash Function

"to hash =*Informal* to make a mess of; mangle"

- Efficiently computable
- Distributes keys uniformly (to minimize collisions)
- Deterministic: **h(k)** is always equal to **h(k)**

Simple uniform hashing:

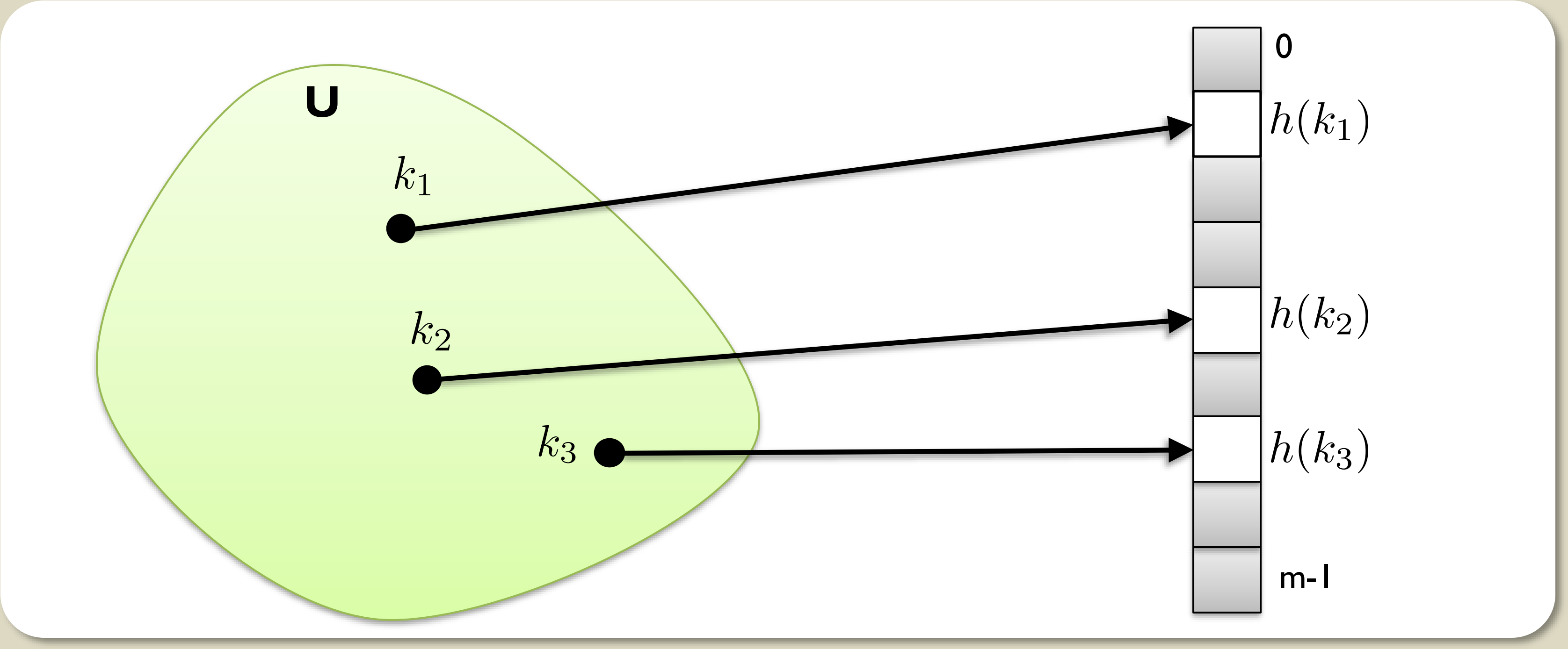
h hashes a new key equally likely to any of the **m** slots independently of where any other has hashed to

Collisions

- Collision: when two items with keys $\mathbf{k}_i$ and $\mathbf{k}_j$ have $h(\mathbf{k}_i) = h(\mathbf{k}_j)$

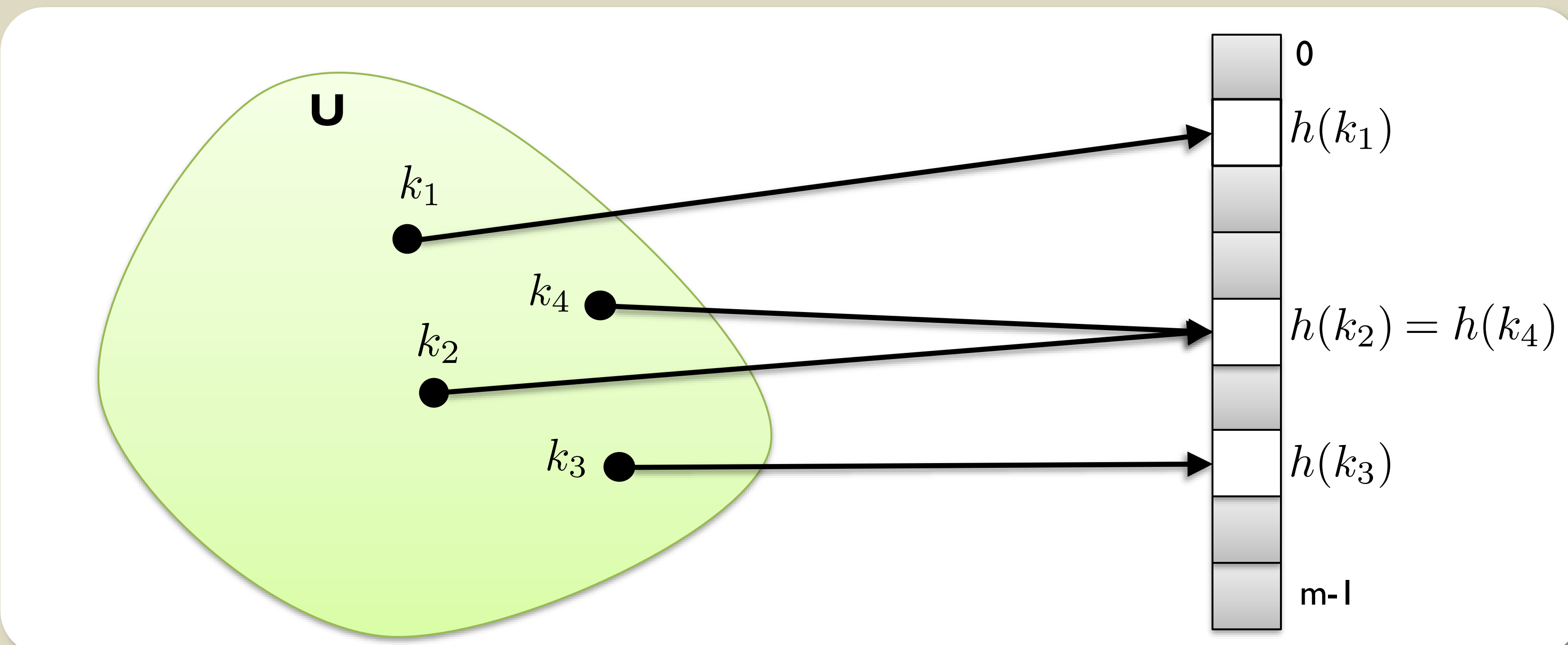
Collisions

- Collision: when two items with keys $\mathbf{k_i}$ and $\mathbf{k_j}$ have $h(\mathbf{k_i}) = h(\mathbf{k_j})$



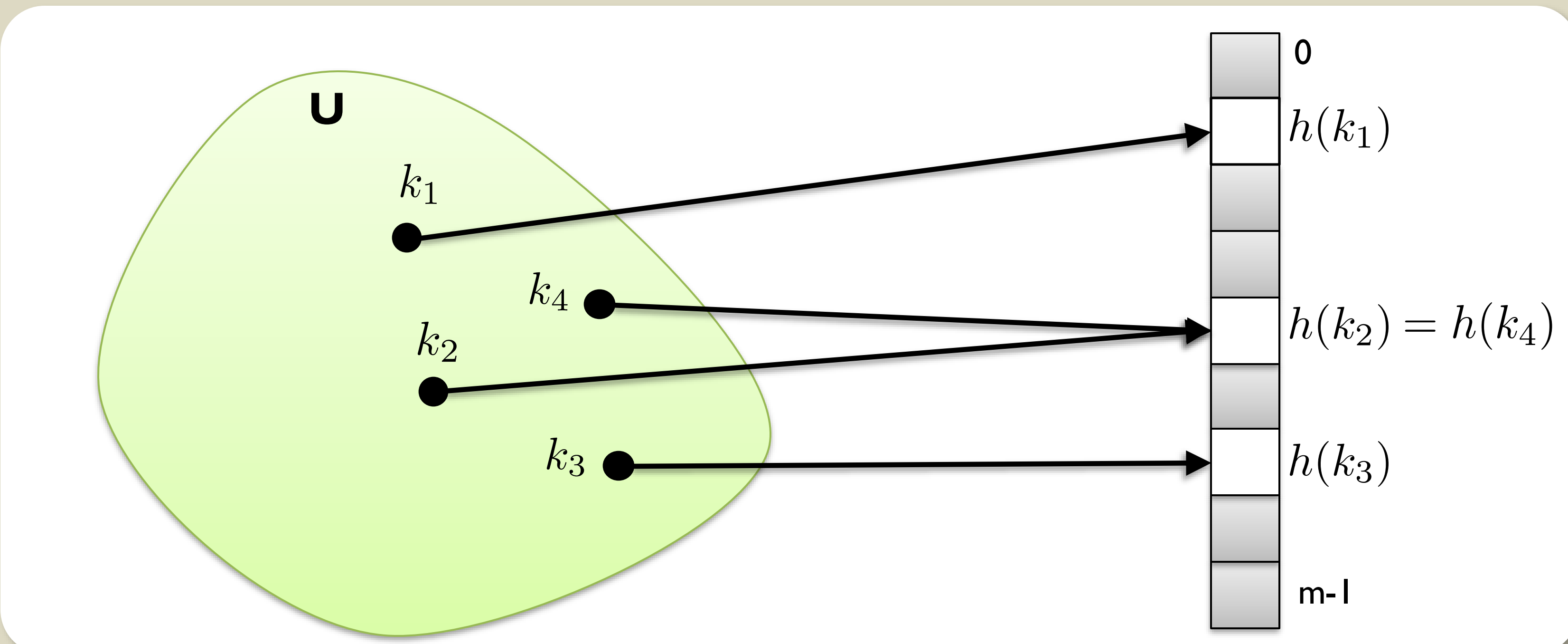
Collisions

- Collision: when two items with keys $\mathbf{k_i}$ and $\mathbf{k_j}$ have $h(\mathbf{k_i}) = h(\mathbf{k_j})$



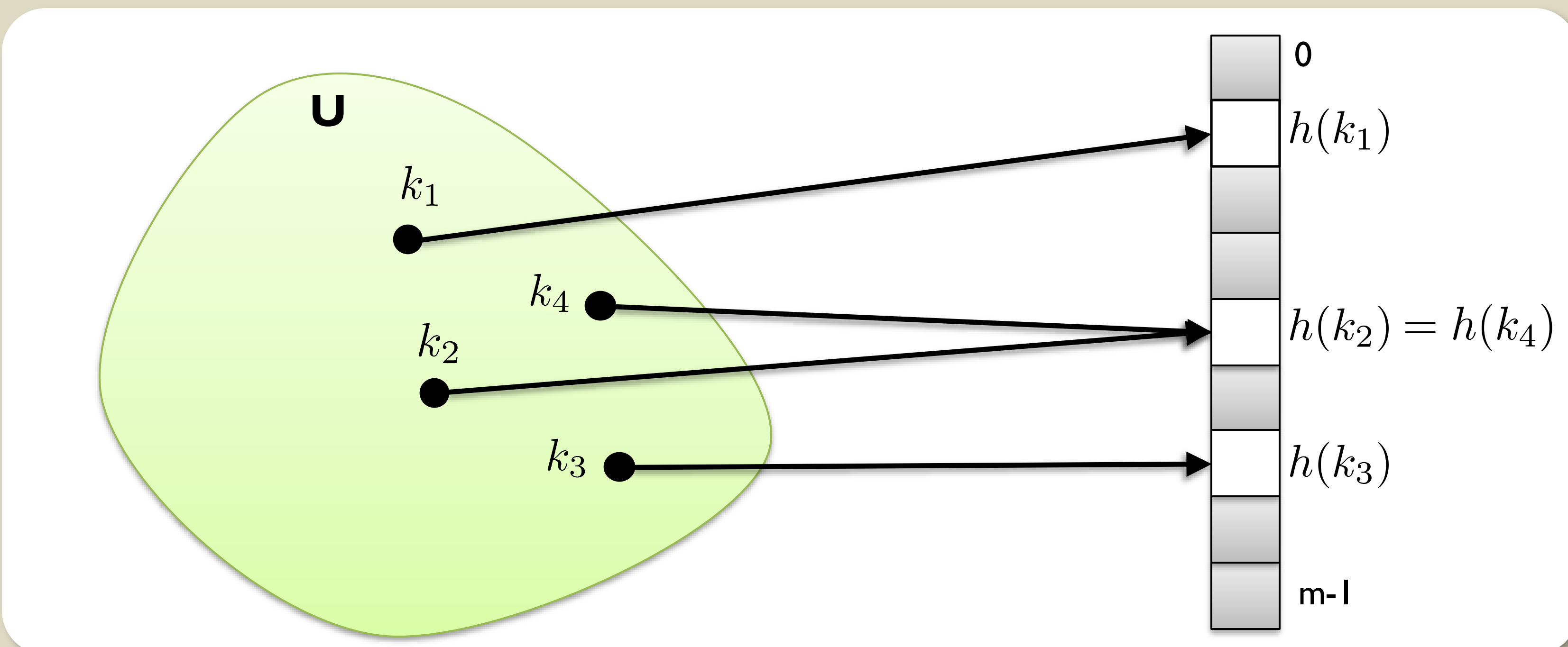
Collisions

- Collision: when two items with keys $\mathbf{k_i}$ and $\mathbf{k_j}$ have $h(\mathbf{k_i}) = h(\mathbf{k_j})$
- How big table do we need to have so as to avoid collisions with high probability?



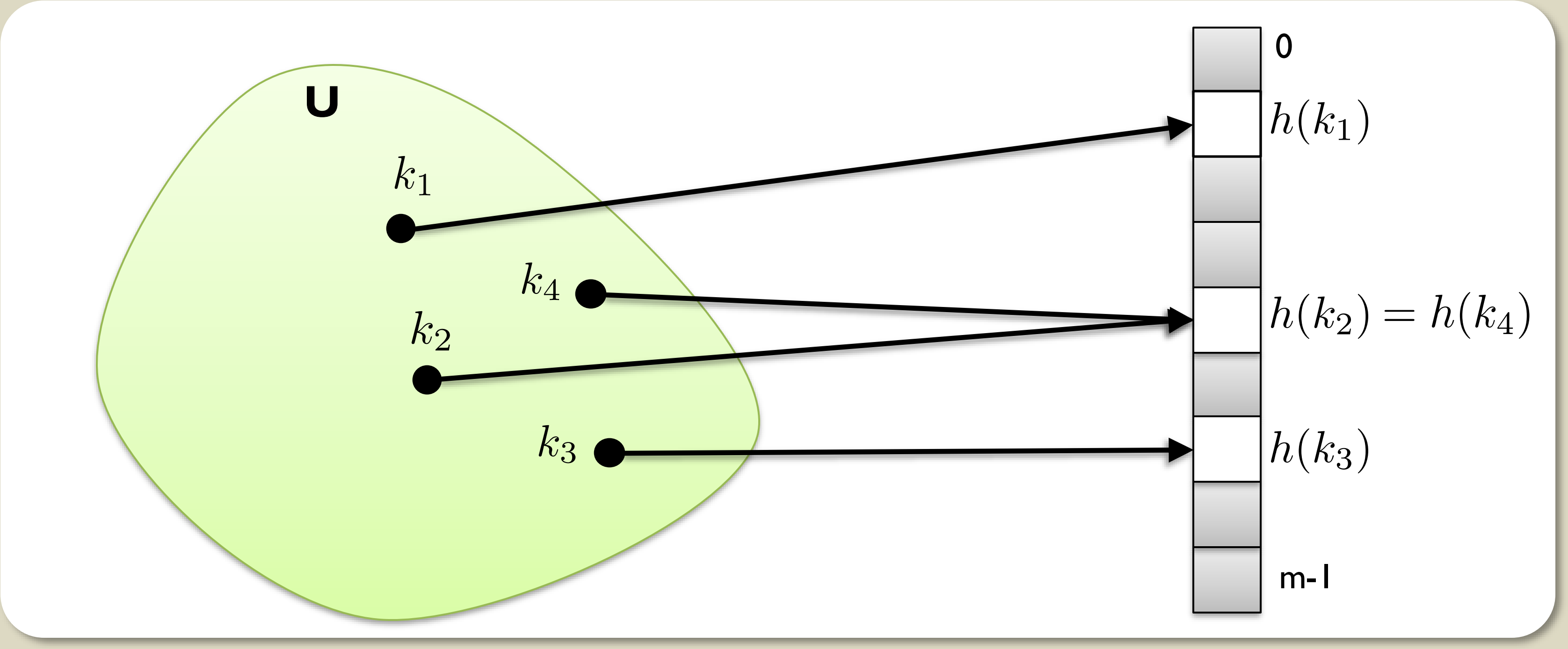
Collisions

- Collision: when two items with keys $\mathbf{k_i}$ and $\mathbf{k_j}$ have $h(\mathbf{k_i}) = h(\mathbf{k_j})$
- How big table do we need to have so as to avoid collisions with high probability?



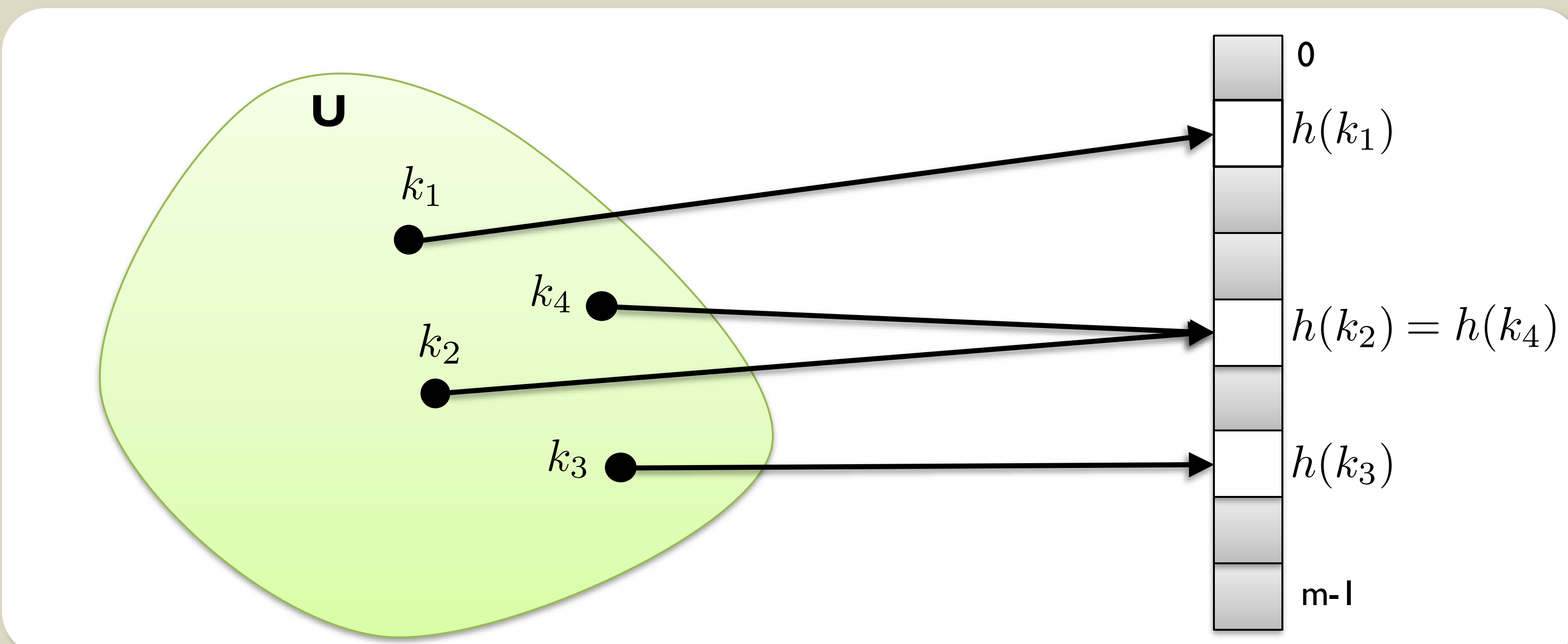
Collisions

- Collision: when two items with keys $\mathbf{k_i}$ and $\mathbf{k_j}$ have $h(\mathbf{k_i}) = h(\mathbf{k_j})$
- How big table do we need to have so as to avoid collisions with high probability?



Collisions

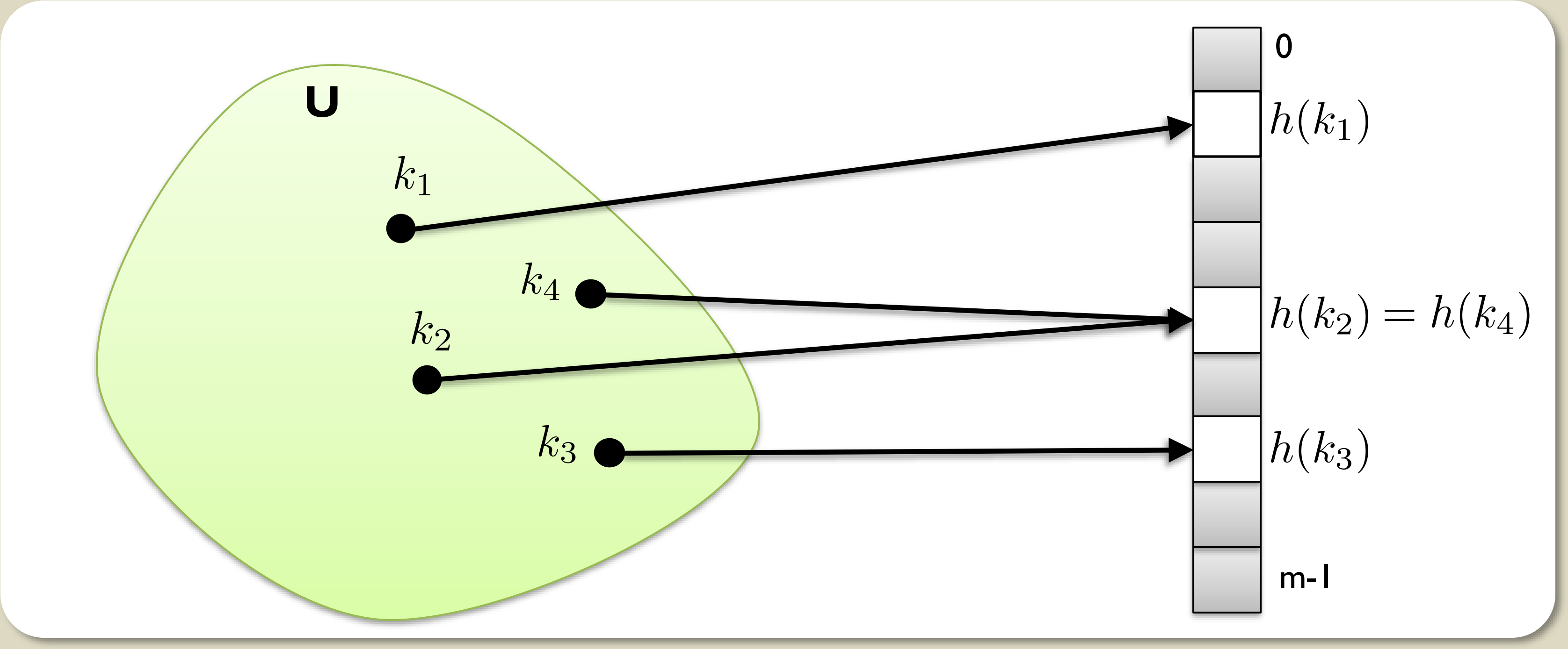
- Collision: when two items with keys $\mathbf{k_i}$ and $\mathbf{k_j}$ have $h(\mathbf{k_i}) = h(\mathbf{k_j})$
- How big table do we need to have so as to avoid collisions with high probability?



Collisions

- Collision: when two items with keys $\mathbf{k_i}$ and $\mathbf{k_j}$ have $h(\mathbf{k_i}) = h(\mathbf{k_j})$
- How big table do we need to have so as to avoid collisions with high probability?

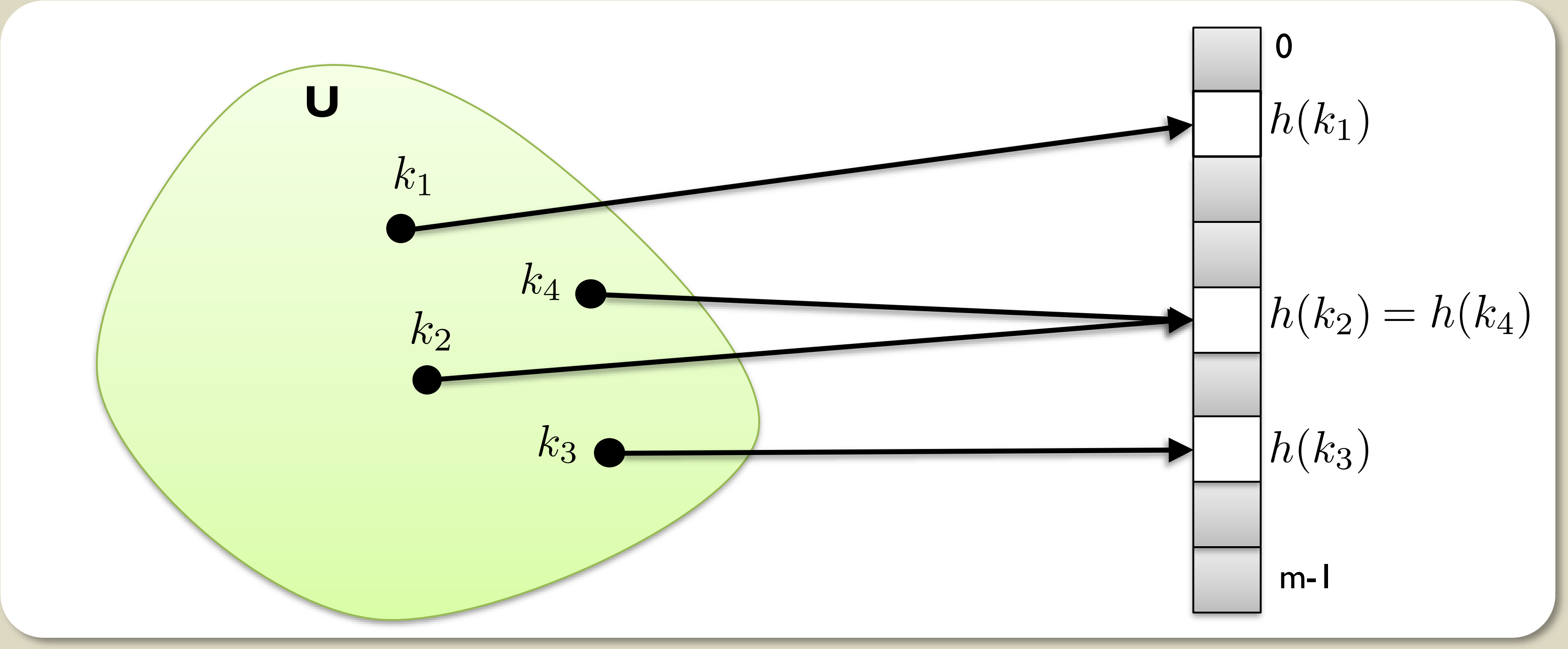
Birthday Lemma says that for **h** to be injective with good probability then we need $\mathbf{m > K^2}$



Collisions

- Collision: when two items with keys $\mathbf{k_i}$ and $\mathbf{k_j}$ have $h(\mathbf{k_i}) = h(\mathbf{k_j})$
- How big table do we need to have so as to avoid collisions with high probability?

Birthday Lemma says that for **h** to be injective with good probability then we need $\mathbf{m > K^2}$ => if library has **10 000** books need array of size $\mathbf{10^8}$

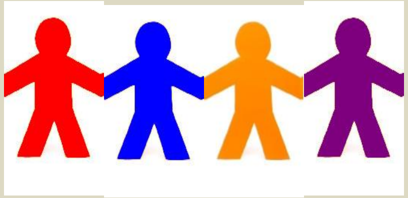


Collisions

You can't avoid them but you can deal with them

Collisions

You can't avoid them but you can deal with them



Collisions

You can't avoid them but you can deal with them

CHAINING:

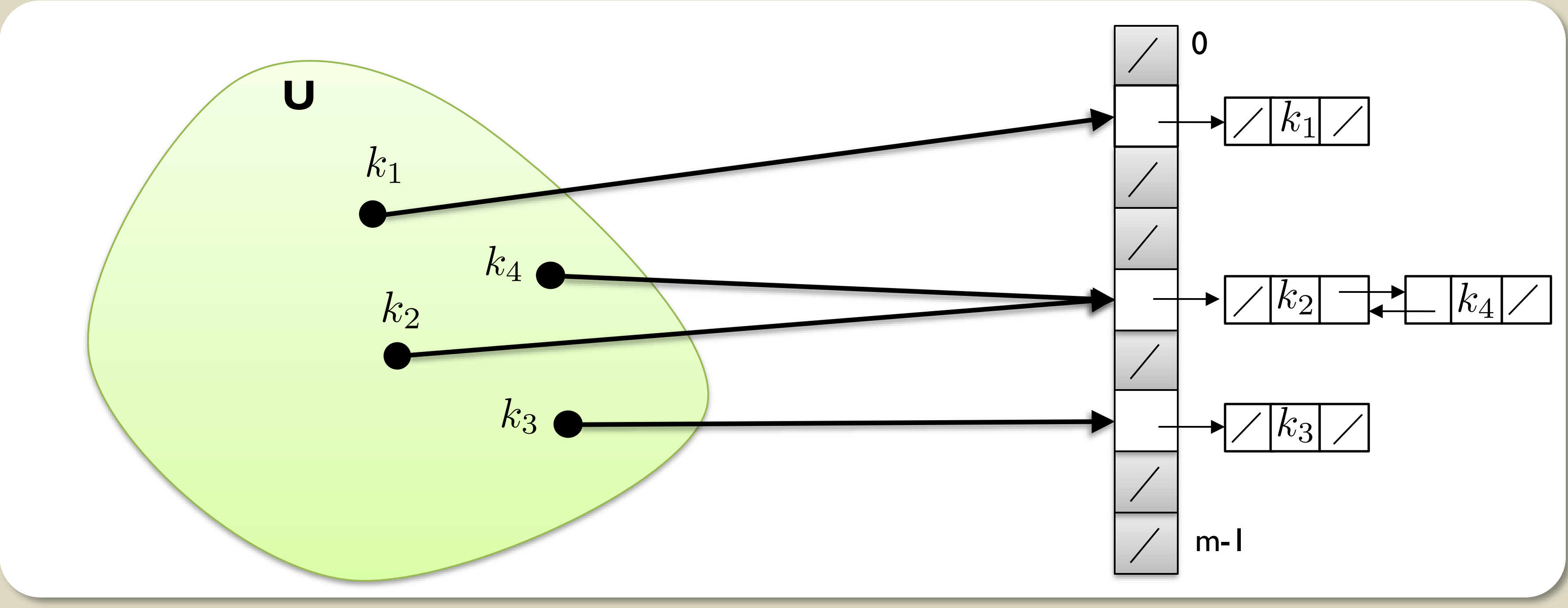
place all elements that hash to the same slot into the same linked list

Collisions

You can't avoid them but you can deal with them

CHAINING:

place all elements that hash to the same slot into the same linked list



Collisions

Chained-Hash-Search(T,k):

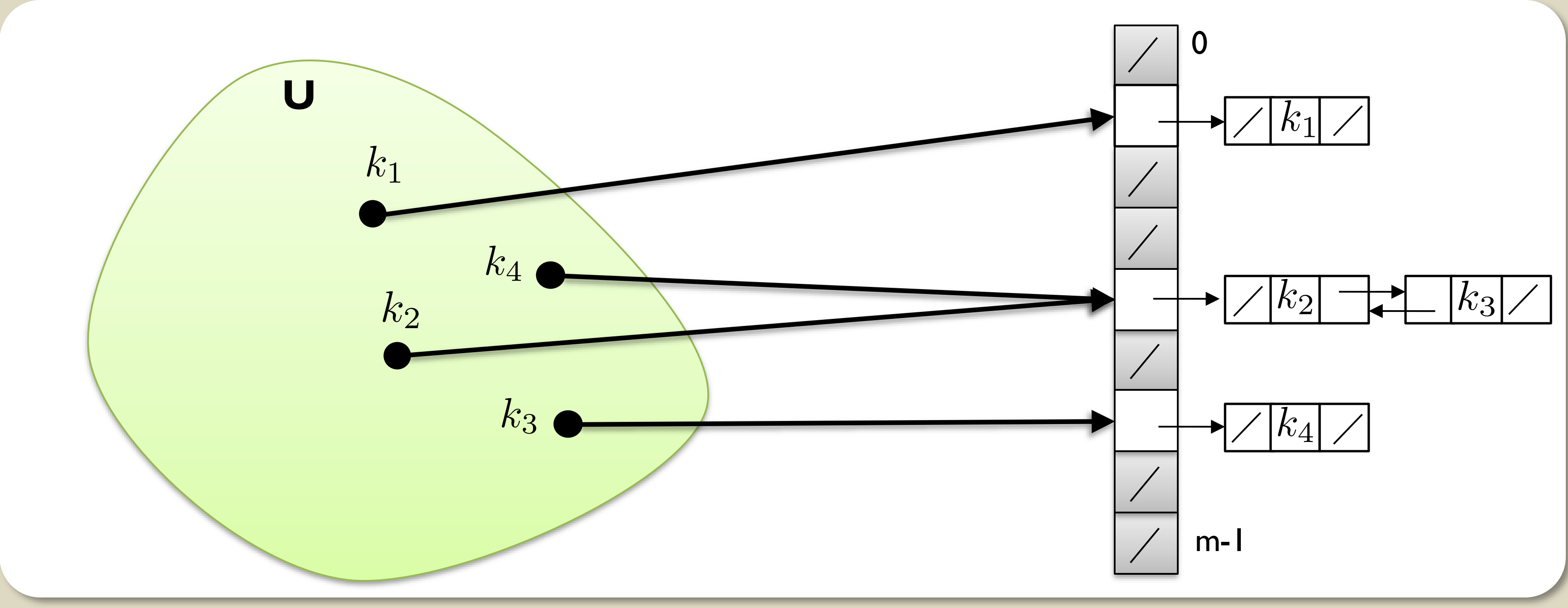
search for an element with key k in list T[h(k)]

Chained-Hash-Insert(T,x):

insert x at the head of list T[h(x.key)]

Chained-Hash-Delete(T,x):

delete x from the list T[h(x.key)]



Collisions

Running time?

Space?

Chained-Hash-Search(T,k):

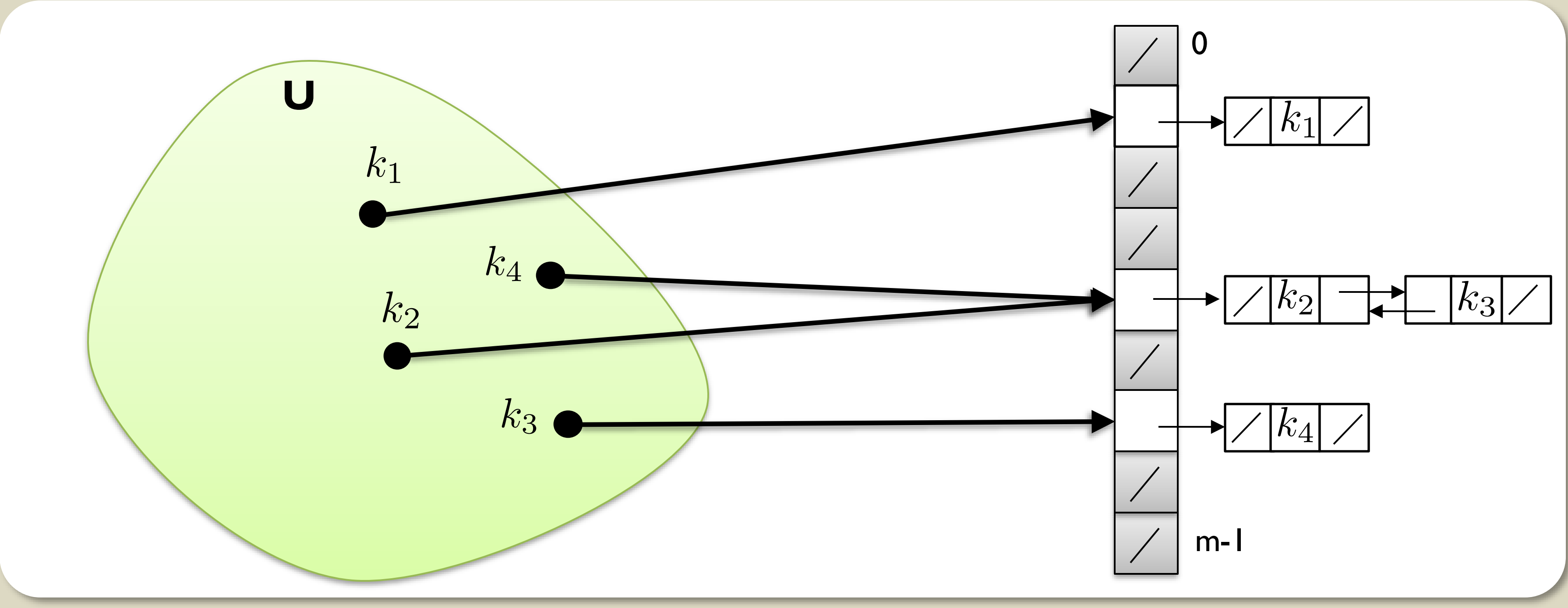
search for an element with key k in list T[h(k)]

Chained-Hash-Insert(T,x):

insert x at the head of list T[h(x.key)]

Chained-Hash-Delete(T,x):

delete x from the list T[h(x.key)]



Collisions

Running time? **O(1)** for insertion, deletion

Space? **O(m+K)**

Chained-Hash-Search(T,k):

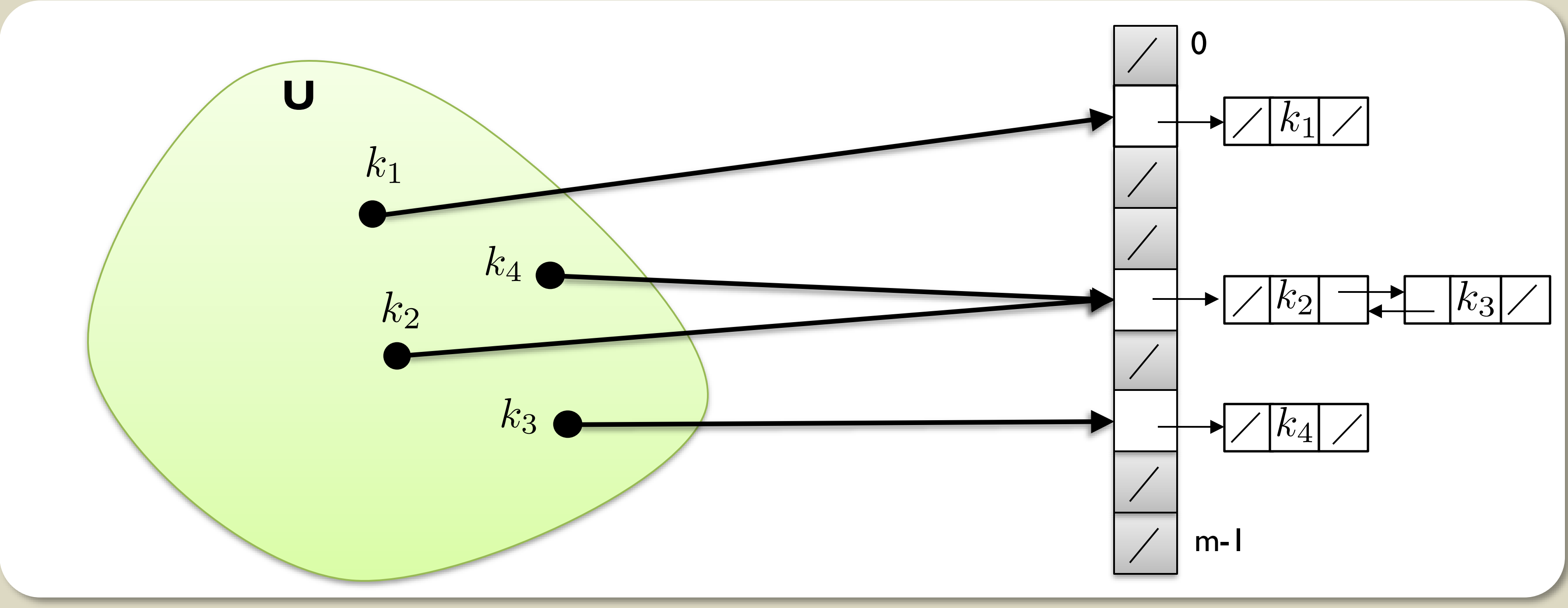
search for an element with key k in list T[h(k)]

Chained-Hash-Insert(T,x):

insert x at the head of list T[h(x.key)]

Chained-Hash-Delete(T,x):

delete x from the list T[h(x.key)]



Collisions

Chained-Hash-Search(T,k):

search for an element with key k in list T[h(k)]

Chained-Hash-Insert(T,x):

insert x at the head of list T[h(x.key)]

Chained-Hash-Delete(T,x):

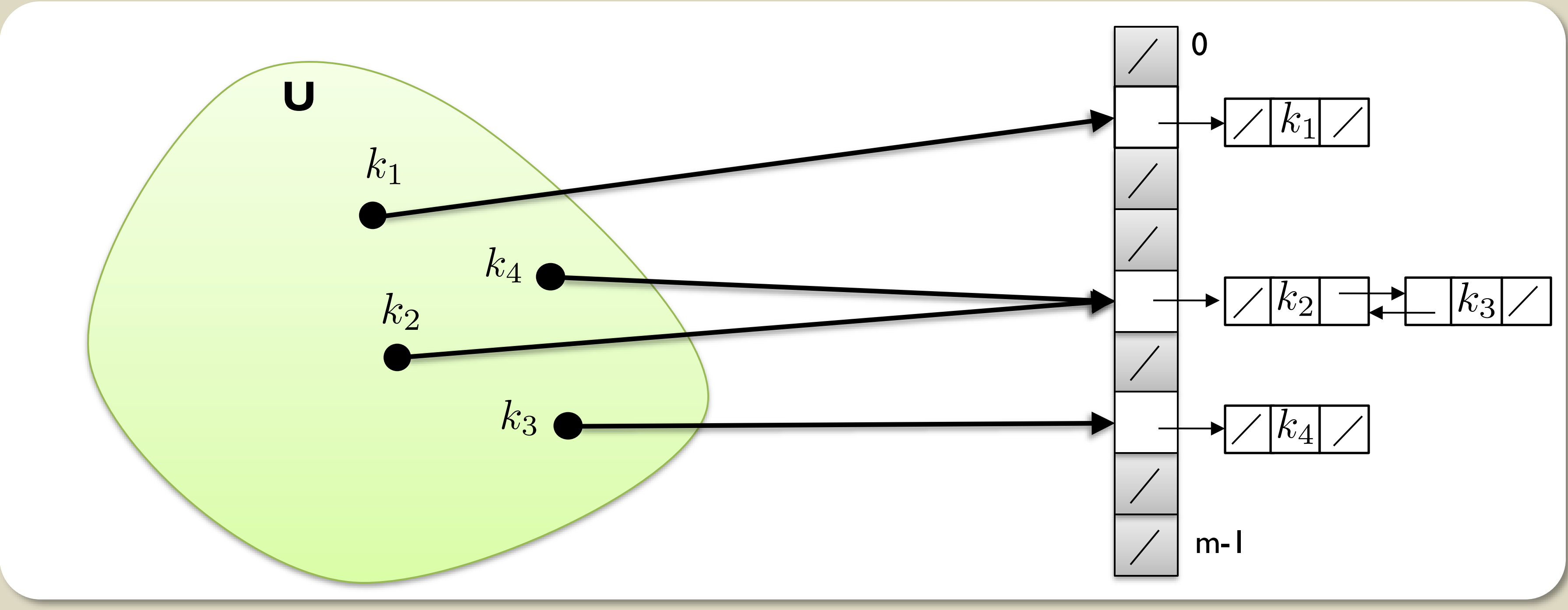
delete x from the list T[h(x.key)]

Running time? **O(1)** for insertion, deletion

Space? **O(m+K)**

Deletion in time O(1) since

- list is doubly linked
- and we are given a pointer to element and not the key



Running Time of Search

Running Time of Search

- Worst case all **n** elements are hashed to same slot
  - Search takes **θ(n)** time in worst case (worse than linked list since we also compute h(k) ☹)
- Analyze average-case behavior
  - We assume we use *simple uniform hashing*

Running Time of Search

- Worst case all **n** elements are hashed to same slot
  - Search takes **$\theta(n)$** time in worst case (worse than linked list since we also compute h(k) ☹)
- Analyze average-case behavior
  - We assume we use *simple uniform hashing*

- Let $\mathbf{n_j}$ denote the length of the list **T[j]**

  - Note that $\mathbf{n = n_0 + n_1 + n_2 + \ldots + n_{m-1}}$

  - And $\mathbf{E[n_j] = Pr[h(k_1) = j] + Pr[h(k_2) = j] + \ldots\ Pr[h(k_n) = j] = \alpha = n/m}$

Running Time of Unsuccessful Search

THEOREM: An unsuccessful search takes expected time **$\theta(1+\alpha)$**

PROOF:

Running Time of Unsuccessful Search

THEOREM: An unsuccessful search takes expected time **$\theta(1+\alpha)$**

PROOF:

- Simple uniform hashing => any key not in the table equally likely to hash to any of the **m** slots

Running Time of Unsuccessful Search

THEOREM: An unsuccessful search takes expected time **$\theta(1+\alpha)$**

PROOF:

- Simple uniform hashing => any key not in the table equally likely to hash to any of the **m** slots

- To search unsuccessfully for any key **k**, need to search till the end of list T[h(k)].
  - This list has expected length $E[n_{h(k)}] = \alpha$
  - Therefore, the expected number of elements examined in an unsuccessful search is α

Running Time of Unsuccessful Search

THEOREM: An unsuccessful search takes expected time $\boldsymbol{\theta(1+\alpha)}$

PROOF:

- Simple uniform hashing => any key not in the table equally likely to hash to any of the **m** slots

- To search unsuccessfully for any key **k**, need to search till the end of list T[h(k)].
  - This list has expected length $E[n_{h(k)}] = \alpha$
  - Therefore, the expected number of elements examined in an unsuccessful search is α

- Adding in the cost for computing the hash function, the total expected time needed is $\Theta(1+\alpha)$

Running Time of Successful Search

- Circumstances are slightly different from unsuccessful search
- The probability that each list is searched is proportional to its length
- (we assume that each element is equally likely to be searched for)

Running Time of Successful Search

THEOREM: A successful search takes expected time **$\theta(1+\alpha)$**

PROOF:

Running Time of Successful Search

THEOREM: A successful search takes expected time **$\theta(1+\alpha)$**

PROOF:

- Let **x** be the element we search for (picked uniformly at random)

Running Time of Successful Search

THEOREM: A successful search takes expected time **$\theta(1+\alpha)$**

PROOF:

- Let **x** be the element we search for (picked uniformly at random)
- The number of elements examined during a successful search for **x** is 1 more than the number of elements that appear before **x** in its list
  - These are the elements inserted after **x** was inserted

Running Time of Successful Search

THEOREM: A successful search takes expected time $\theta(1+\alpha)$

PROOF:

- Let **x** be the element we search for (picked uniformly at random)
- The number of elements examined during a successful search for **x** is 1 more than the number of elements that appear before **x** in its list
  - These are the elements inserted after **x** was inserted
- So we need to find the average, over the **n** elements **x** in the table, of how many elements were inserted into **x**'s list after **x** was inserted

Running Time of Successful Search

THEOREM: A successful search takes expected time **$\theta(1+\alpha)$**

PROOF:

- For i = 1, 2, …, n, let **x_i** be the i'th element inserted into the table and let **k_i = key[x_i]**

Running Time of Successful Search

THEOREM: A successful search takes expected time $\theta(1+\alpha)$

PROOF:

- For i = 1, 2, …, n, let x_i be the i'th element inserted into the table and let $k_i = key[x_i]$
- For all **i** and **j**, define indicator variable
  - $X_{ij} = I\{h(k_i) = h(k_j)\}$
  - Simple uniform hashing => $E[X_{ij}] = 1/m$

Running Time of Successful Search

THEOREM: A successful search takes expected time $\theta(1+\alpha)$

PROOF:

- For i = 1, 2, …, n, let x_i be the i'th element inserted into the table and let k_i = key[x_i]
- For all **i** and **j**, define indicator variable
  - $X_{ij} = I\{h(k_i) = h(k_j)\}$
  - Simple uniform hashing => $E[X_{ij}] = 1/m$
- Expected number of elements examined in a successful search is

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

Running Time of Successful Search

THEOREM: A successful search takes expected time **$\theta(1+\alpha)$**

PROOF:

- For i = 1, 2, …, n, let $\mathbf{x_i}$ be the i'th element inserted into the table and let $\mathbf{k_i = key[x_i]}$
- For all **i** and **j**, define indicator variable
  - $\mathbf{X_{ij} = I\{h(k_i) = h(k_j)\}}$
  - Simple uniform hashing => $\mathbf{E[X_{ij}] = 1/m}$
- Expected number of elements examined in a successful search is

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

Running Time of Successful Search

THEOREM: A successful search takes expected time **$\theta(1+\alpha)$**

PROOF:

- For i = 1, 2, …, n, let $\mathbf{x_i}$ be the i'th element inserted into the table and let $\mathbf{k_i} = \mathbf{key[x_i]}$
- For all **i** and **j**, define indicator variable
  - $\mathbf{X_{ij} = I\{h(k_i) = h(k_j)\}}$
  - Simple uniform hashing => $\mathbf{E[X_{ij}] = 1/m}$
- Expected number of elements examined in a successful search is

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

Running Time of Successful Search

THEOREM: A successful search takes expected time **$\theta(1+\alpha)$**

PROOF:

- For i = 1, 2, …, n, let $\mathbf{x_i}$ be the I’th element inserted into the table and let $\mathbf{k_i = key[x_i]}$
- For all **i** and **j**, define indicator variable
  - $\mathbf{X_{ij} = I\{h(k_i) = h(k_j)\}}$
  - Simple uniform hashing => $\mathbf{E[X_{ij}] = 1/m}$
- Expected number of elements examined in a successful search is

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$= \cdots = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

Consequence of Analysis

- Recall that **$\alpha = n/m$**

- So if we choose the size our hash table to be proportional to the number of elements stored.
  - **$m = \Theta(n)$**

- Then insertion, deletion **$O(1)$** time and search expected **$O(1)$** time

Examples of Hash Functions

- Big area of research
- Depends on data distribution and other properties
- We give two basic examples

Examples of Hash Functions

Division method

$h(k) = k \bmod m$

m often selected to be a prime not too close to a power of 2

Examples of Hash Functions

Division method

h(k) = *k* mod *m*

m often selected to be a prime not too close to a power of 2

Multiplicative method

h(k) = floor[m * FractionalPartOf(Ak)]

Knut suggests to chose $A \approx (\sqrt{5} - 1)/2$

Summary

- Probabilistic analysis (the hiring problem)
  - Random indicator variables
  - Linearity of expectation:

    E[aX + bY] = aE[X] + bE[Y]

    *holds even if **X** and **Y** are dependent*

- Hash tables
  - Very practical method with fast insertion, deletion, and search
  - Performance depends on choice of hash function
  - Resolve conflicts by for example using chaining