Hashing

Thomas Schwarz, SJ

Dictionary

- ADS for key-value pairs
  - CRUD operations:
    - Create
    - Read
    - Update
    - Delete
  - Does not assume nor support ordering of keys

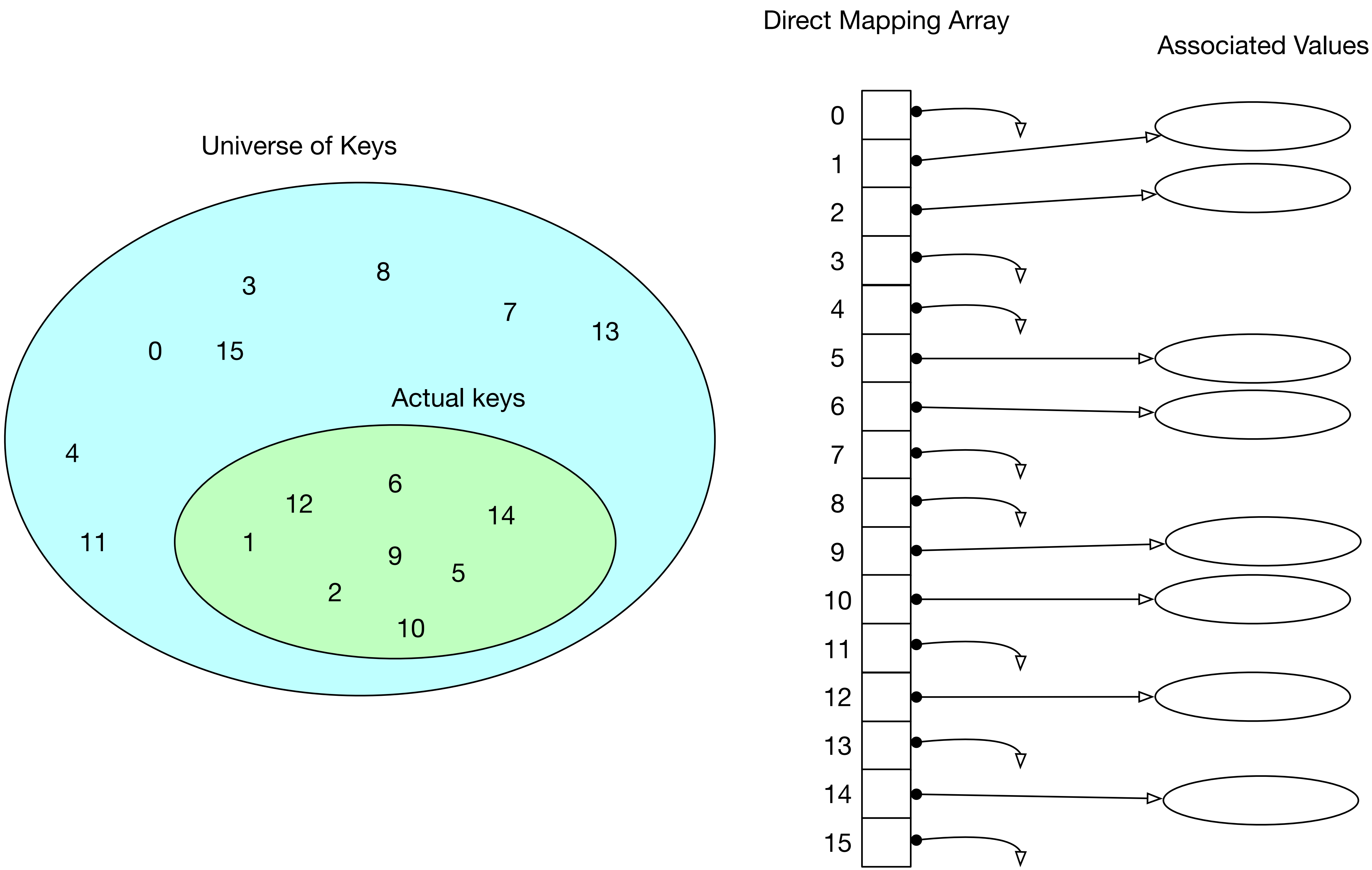
Dictionary

- Example:
  - A compiler takes a variable name (= key)
  - and associates various data such as type etc. (value)

Direct Addressing

- If the key space is small
    - Use Direct Addressing
        - Array for all possible key values with pointers to values
        - Null-pointers (None) if key not in the dictionary

Direct Addressing



Direct Addressing

- Direct addressing:
  - Number of actual keys needs to be close to the number of possible keys
  - Keys need to be convertible to indices

Direct Addressing

- Direct addressing:
  - Number of actual keys needs to be close to the number of possible keys
  - Keys need to be convertible to indices

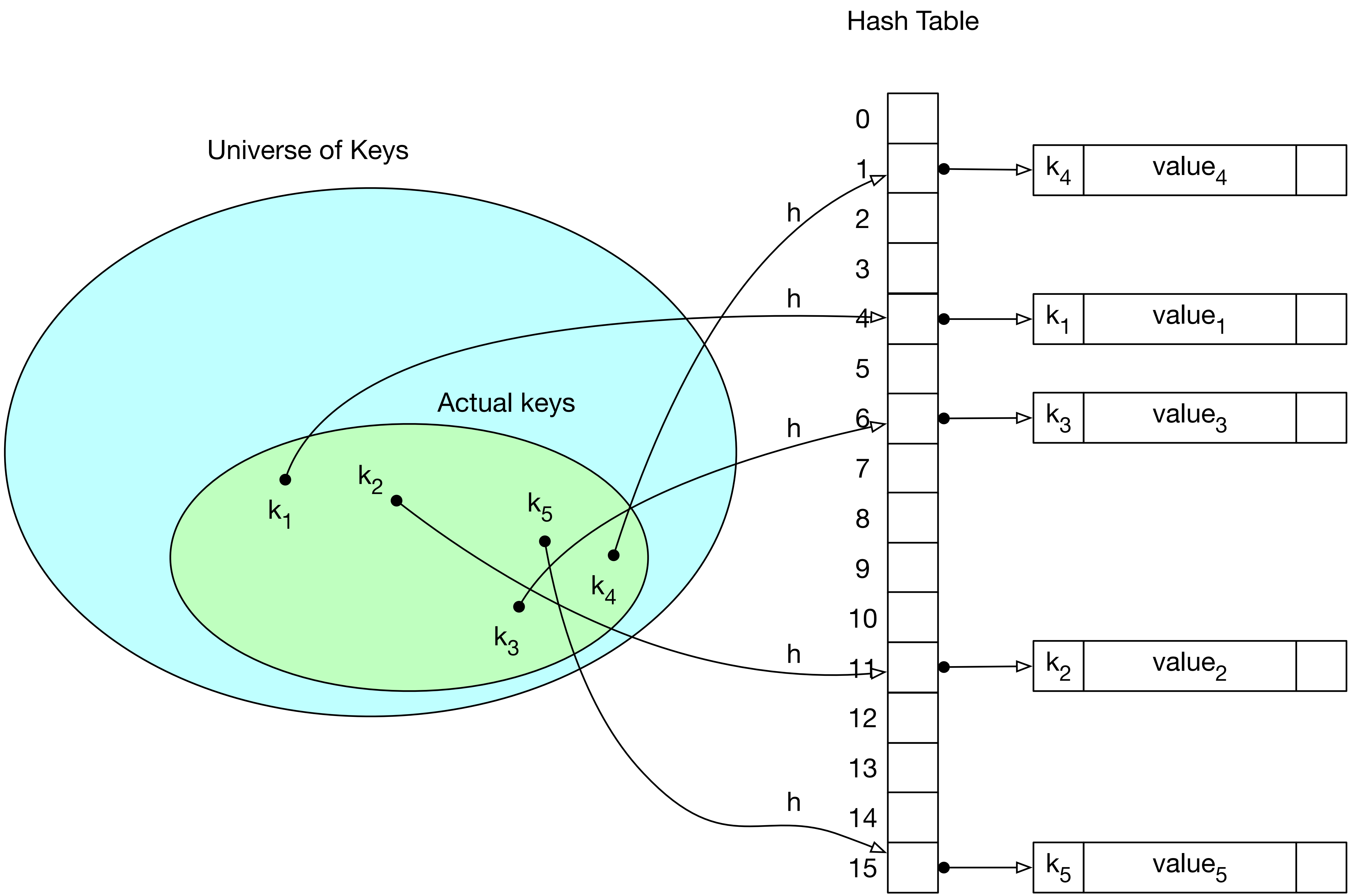
Direct Addressing

- Variants: The value can be stored directly in the array
  - E.g.: When the value has fixed length

Hash Tables

- If the universe U of keys is large
  - Table with $|U|$ entries is too big
  - And most of its entries would be Nones
- Use a *hash* function
  - $h : U \longrightarrow \{0,1,\ldots,m-1\}$
  - with few *collisions*
    - A collision are two elements of U that map to the same number
    - $u_1, u_2 \in U, u_1 \neq u_2, h(u_1) = h(u_2)$

Hash Tables

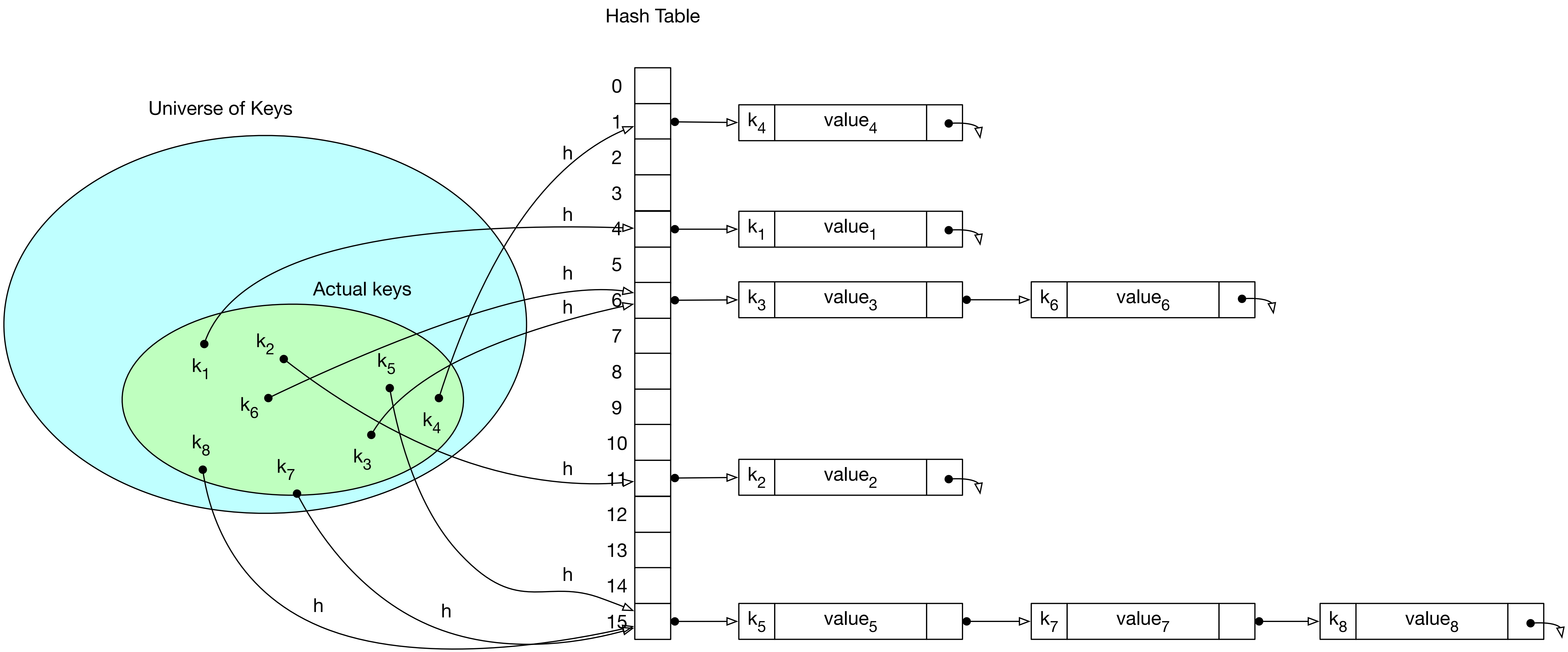


Hash Table
Universe of Keys
Actual keys
k1
k2
k3
k4
k5
h
h
h
h
h
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
k4
value4
k1
value1
k3
value3
k2
value2
k5
value5

Hash Tables

- Collisions happen and they must be resolved
    - Chaining:
        - create a linked list of key-value pairs

Hash Tables



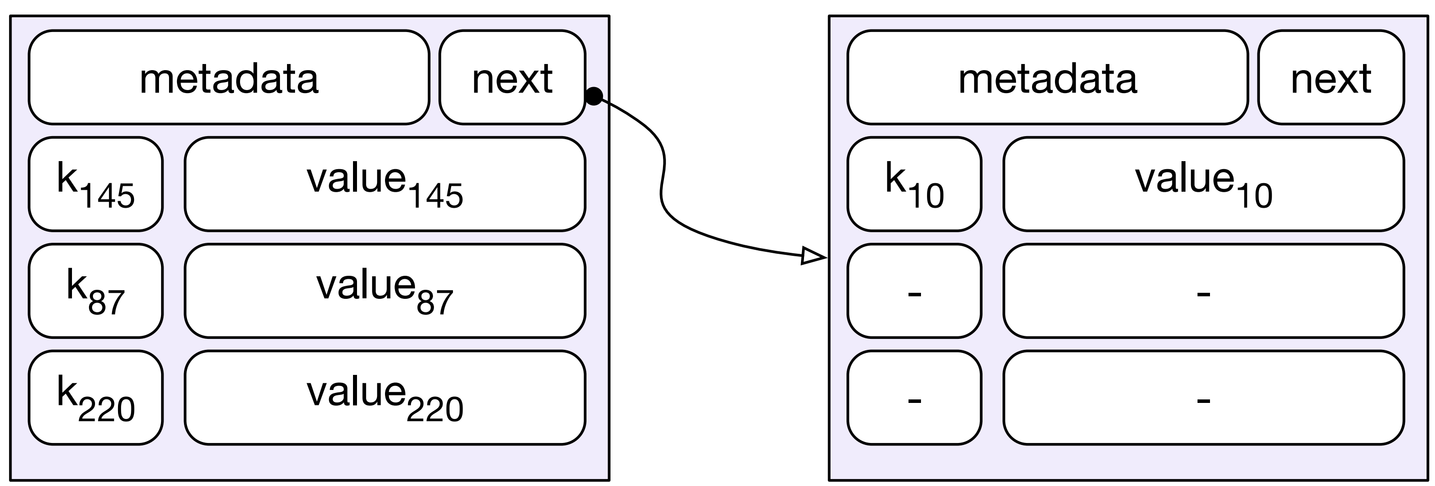
Hash Table
Universe of Keys
Actual keys
k_1
k_2
k_3
k_4
k_5
k_6
k_7
k_8
h
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
k_4 value$_4$
k_1 value$_1$
k_3 value$_3$
k_6 value$_6$
k_2 value$_2$
k_5 value$_5$
k_7 value$_7$
k_8 value$_8$

Hash Tables

- Bucketing
  - A linked list is not necessarily the best way to store key-value pairs
  - If the hash table is large, the data will be stored in the pages of a storage system
  - Can have buckets with a given maximum capacity
    - However, we might need to have overflow buckets

Hash Tables

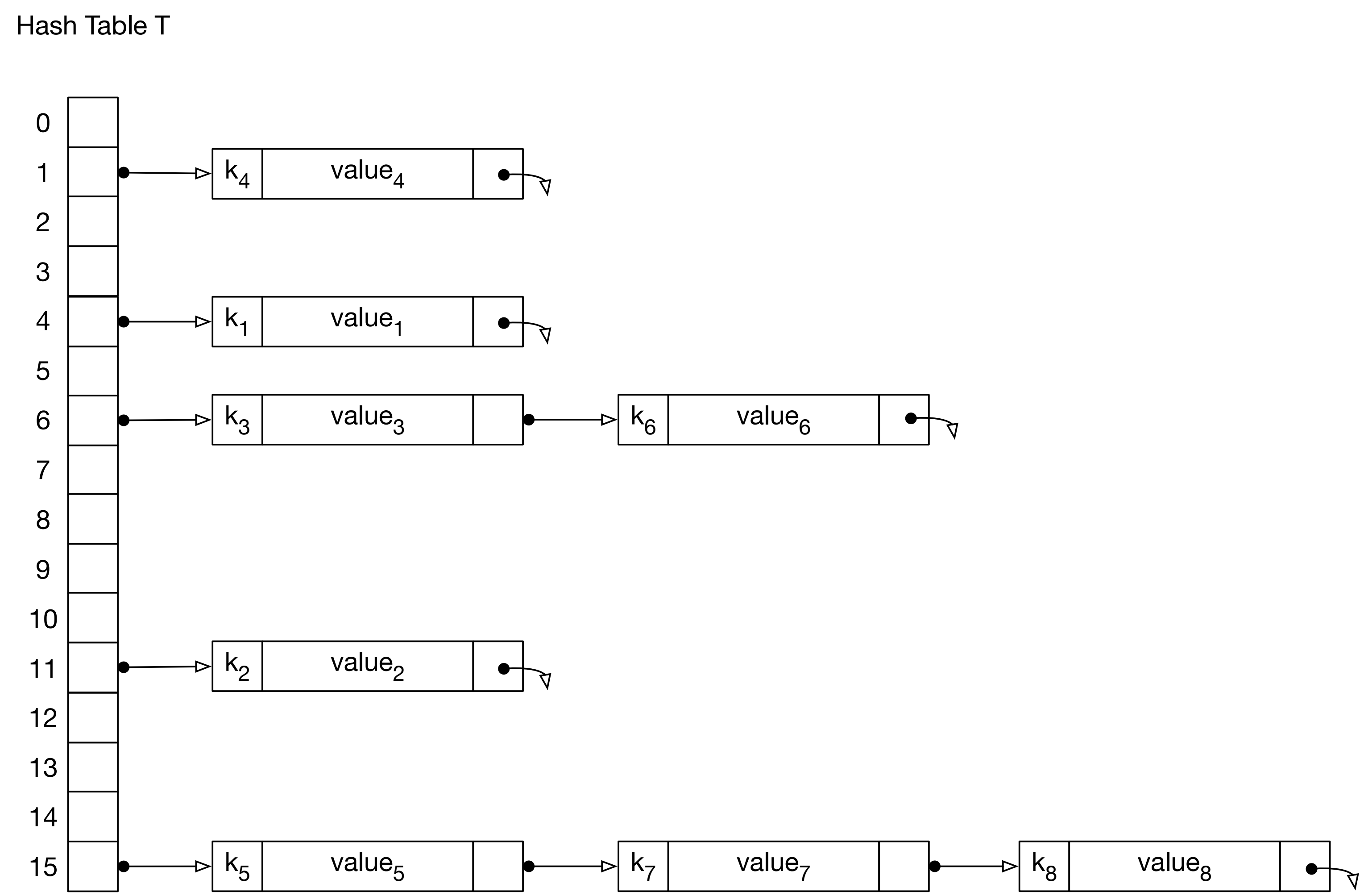
- A potential design for buckets:
  - Each bucket has a next pointer to an overflow area
  - And in this case a fixed capacity to store key-value pairs



- Here: a full bucket with one overflow record

Performance of Hashing with Chaining

- Vocabulary:
  - A hash table T has m slots
  - with n records.
  - Its load factor is $\alpha = n/m$.



$m = 16$
$n = 8$
$\alpha = \frac{1}{2}$

Performance of Hashing with Chaining

- Worst performance:
  - The hash function maps all record keys to the same slot
  - Finding a key-value pair then takes
    - n accesses to a key-value pair if the record is not there
    - On average $(n+1)/2$ accesses if the record is there
      - because

$$(1+2+\ldots+n)/n = \frac{(n+1)n}{2n} = \frac{n+1}{2}$$

Performance of Hashing with Chaining

- Why would this happen:
  - The hash function is bad
    - This happens if people make up their own hash functions
  - The data is cooked
    - "Adversary model": Evaluate algorithms and ADS by finding the worst possible instance of data
    - Someone controls the input and is attacking your system
  - Bad luck
    - Murphy's law: If something bad can happen, it will happen eventually

Performance of Hashing with Chaining

- Average performance analysis:
  - Assume that a hash function is equally likely to send a record to a certain slot
  - This can be *de facto* guaranteed with cryptographically secure hash functions (see below)

Performance of Hashing with Chaining

- Call n_i the number of records (= key-value pairs) that are hashed to slot i ($i \in \{0,1,\ldots,n-1\}$)
- Then $n_0 + n_1 + \ldots n_{m-2} + n_{m-1} = n$
- Expected number of records accessed for an **unsuccessful search**:
  - Equal to the length of the chain, i.e. to n_i
  - On average: $\dfrac{n_0 + n_1 + n_2 + \ldots + n_{m-2} + n_{m-1}}{m} = \dfrac{n}{m} = \alpha$
  - Total expected work: Need to calculate the hash function etc.
    - $\Theta(1+\alpha)$ (the one because α can be zero.)

Performance of Hashing with Chaining

- Successful search:
  - In a list of r records, we access on average $\frac{r+1}{2}$ records
  - If each record **<u>were</u>** in a random slot:
    - Average number of records accessed during a successful search is therefore
    - $\frac{\frac{n_0+1}{2}+\frac{n_1+1}{2}+\ldots\frac{n_{m-2+1}}{2}+\frac{n_{m-1}+1}{2}}{m}$
    - $=\frac{n_0+n_1+\ldots+n_{m-2}+n_{m-1}+m}{2m}$
    - $=\frac{n+m}{2m}=\frac{1}{2}\alpha+1=\Theta(1+\alpha)$

Performance of Hashing with Chaining

- Successful search:
  - But records are more likely to be in full slots
  - Therefore, this analysis is **false**
    - Probability that two keys are in the same slot is $\frac{1}{m}$
    - A search for a record visits exactly those records that:
      - Are in the same slot
      - And have been inserted before

Performance of Hashing with Chaining

- Order all records $[k_0, v_0], [k_1, v_1], \ldots, [k_{n-1}, v_{n-1}]$ by insertion
  - Then search for k_i touches all records with $k_0, k_1, \ldots, k_{i-1}$
    - But only if they are inserted into the same slot
      - which happens with probability $\frac{1}{m}$
  - Therefore:
    - Search for record i looks at $\frac{i}{m}$ records plus itself

Performance of Hashing with Chaining

- Search for record i looks at $\dfrac{i}{m}$ records plus itself
- On average:

$$\frac{1}{n}\left(\sum_{i=0}^{n-1}(\frac{i}{m}+1)\right)$$

$$= 1 + \frac{1}{nm}\sum_{i=0}^{n-1} i$$

$$= 1 + \frac{n(n-1)}{2nm}$$

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

Hash Functions

- A good hash function:
  - each key is equally likely to hash to any of the m slots
  - independently where any other keys are hashed to
- Usually cannot be ascertained:
  - We do not know enough about the distribution of keys

Hash Functions

- Example:
  - Assume that keys are random number between 0 and 1
  - Good hash function is:
    - $h(k) = \lfloor k \cdot m \rfloor$

```
import random

m=5

def hash(u):
    return int(u*m)
```

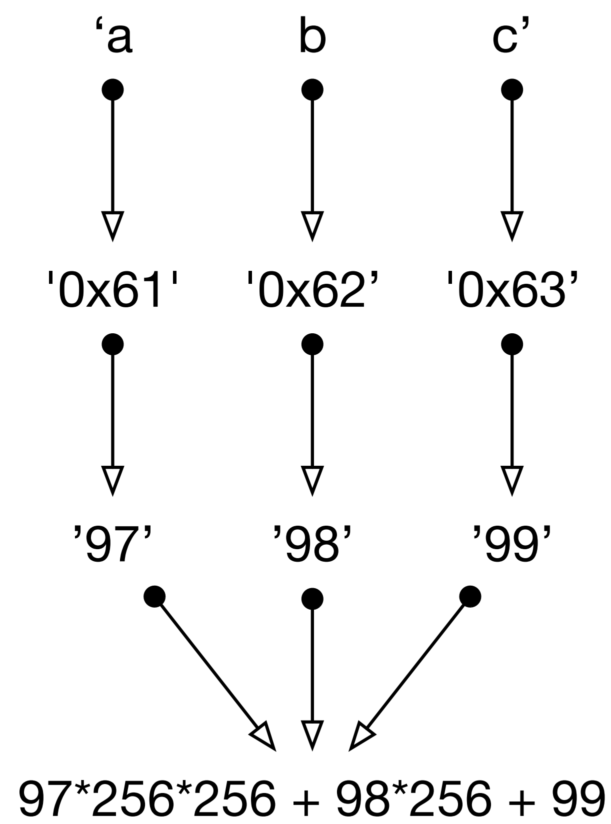
Hash Functions

- Interpreting keys as natural numbers
  - Many hash functions work on natural numbers
  - Need to translate to integers:
    - Example:
      - For strings:
        - convert encoding to numerical representation

```
def str_to_int(astring):
    result = 0
    for letter in astring:
        result = ord(letter) + 256*result
    return result
```

Hash Functions

- Example




```
def str_to_int(astring):
    result = 0
    for letter in astring:
        result = ord(letter) + 256*result
    return result
```

Hash Functions

- Caution:
    - The transformation and the hash function combination can have weird effects
        - E.g. A string obtained by swapping to letters might have the same hash
            - Which could be useful or could be very bad

Hash Functions

- Simple hash functions:
  - Division method:
    - Convert keys to integers
    - Then hash to the integer obtained as remainder by division with m

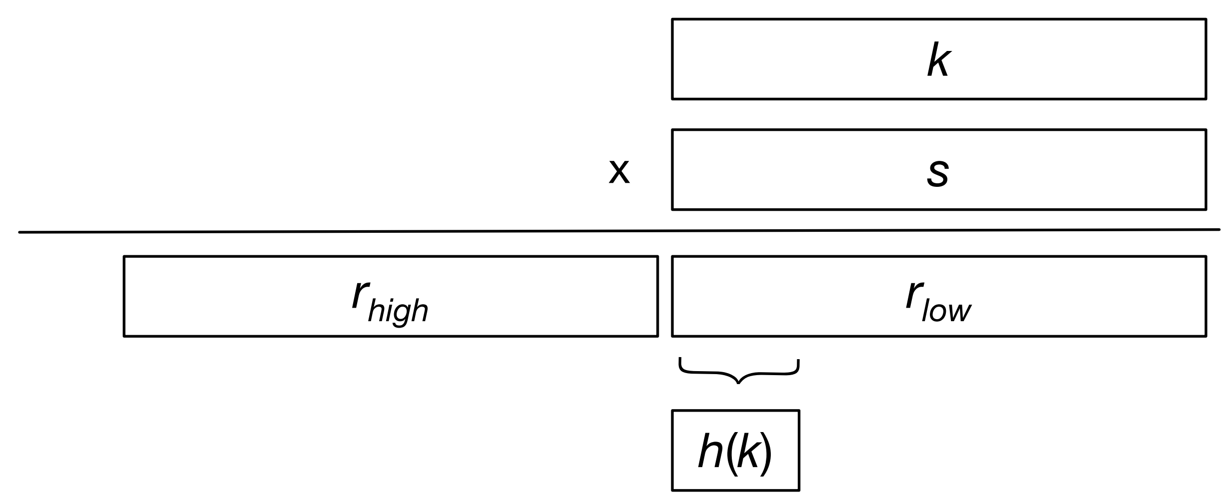
```
def hash(key):
   return key_2_int(key)%m
```

Hash Functions

- Division method:
  - If m is a power of 2:
    - Hash is the last bits
      - Which is usually bad
  - If $m = 2^p - 1$ and keys are strings:
    - swapping two letters does not change the hash value
  - Better experience:
    - Primes close to a power of 2

Hash Functions

- Multiplication Method:
  - Key k is a *l* bit value
  - Use a constant s of size *l* bits
  - Multiply: ks, a 2*l* bit value
  - Select hash as upper bits of the lower half



Hash Functions

- How to select s:
  - D. Knuth proposes to use the first / bits of $\frac{\sqrt{5}-1}{2}$
  -

Hash Functions

- Example:
  - 32-bit keys
  - $s = \lfloor(\sqrt{5} - 1)/2\rfloor$
  - Extract 14 bits:
    - Shift right by 18 (14+18 = 32)
    - Then mask with 14 ones: b11 1111 1111 1111 = 0x3fff

```
s = int((math.sqrt(5)-1)/2 * 2**32)
def hash(x):
    return (s*x >> 18) & 0x3fff
```

Hash Functions

- Cryptographically secure hash functions:
  - Hash functions have applications in security
    - Instead of storing a password, store the hash of a password together with the user name
      - "user_name", h(pass_word)
    - When user enters the password:
      - System calculates the hash of the entered password
      - And compares with the hash

Hash Functions

- Cryptographically secure hash functions:
  - Generate long hashes (224 - 512 bits)
  - If an attacker steals the user database:
    - Attacker has only the hash, but not the password
- Cryptographically secure hash function h:
  - Impossible to calculate x from $h(x)$

Hash Functions

- This is why you should not choose words in a language as password: "peaches"
  - Attacker can try out
    - All words in English (~200,000),
    - All words in Hindi ShabdSagar (~93,000 - 250,000)

Hash Functions

- Secure hash functions are the result of competitions and public scrutiny
  - Provide pre-image resistance:
    - Impossible to find x from $h(x)$
  - Provide collision resistance:
    - Impossible to find x and y such that $h(x) = h(y)$
  - Certified by NIST and similar institutions
    - SHA-3 (NIST)
    - Blake3 (latest considered to be safe)

Hash Functions

- Should you use cryptographically secure hash functions?
  - If your data cannot be generated by an adversary
  - If you can live with small inadequacies
    - Not necessary
- Otherwise:
  - Extract as many bits as needed from a cryptographically secure hash function
  - Pay the performance costs

Open Addressing Hashing

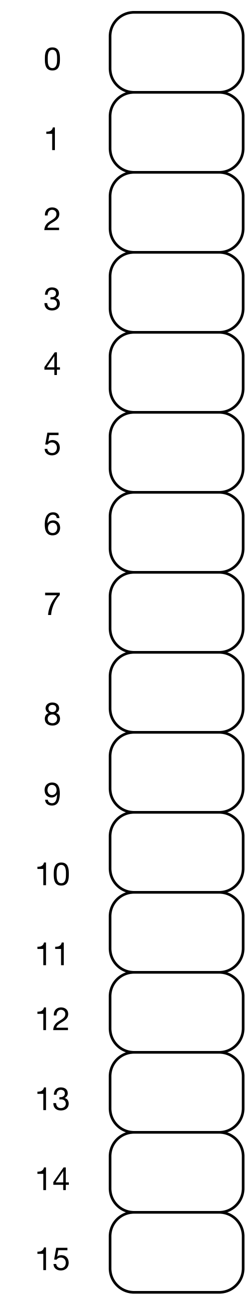
- Idea: Records (= key-value pair) are stored in the hash table itself
  - Collisions are resolved by storing a record elsewhere

Open Addressing Hashing

- Linear probing:
  - If a slot is occupied, go to the next slot

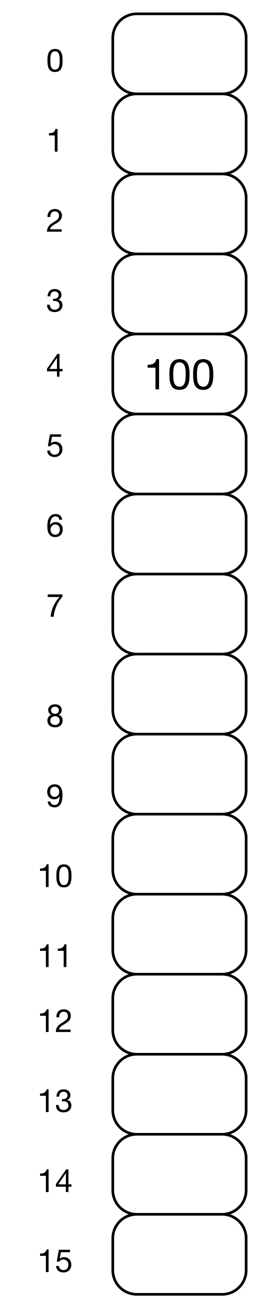
Open Addressing Hashing

- Linear Probing Example:
  - 16 slots
  - Hash function %16



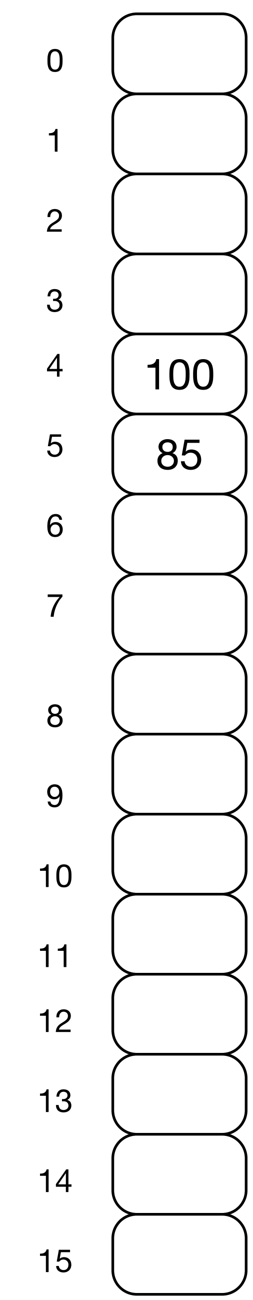
Open Addressing Hashing

- Linear Probing Example:
  - 16 slots
  - Hash function %16
  - Insert 100
  - 100%16 = 4
    - Insert into slot 4



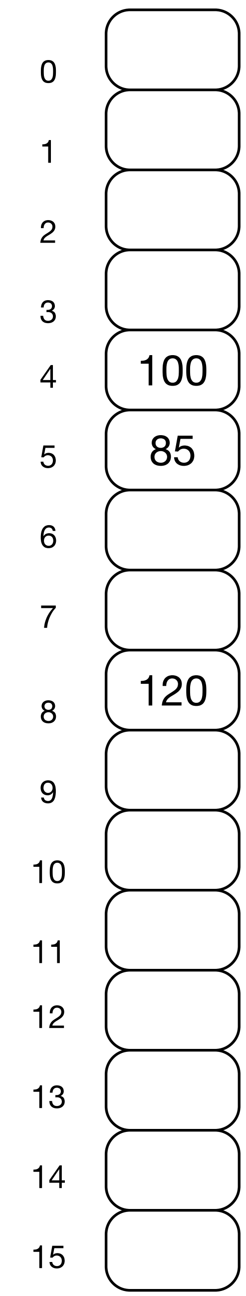
Open Addressing Hashing

- Insert 85
- 85%16 = 5
- Insert into slot 5



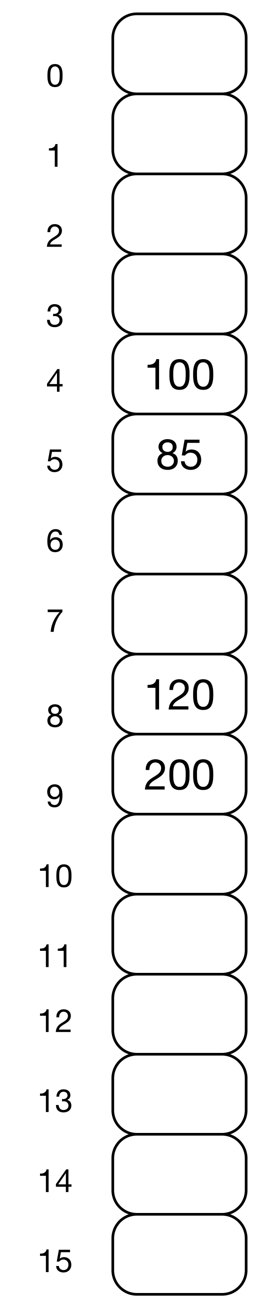
Open Addressing Hashing

- Insert 120
  - 120%16 = 8



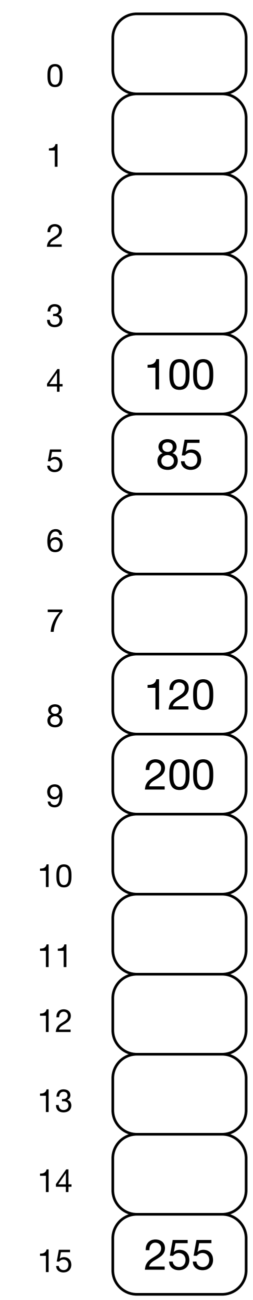
Open Addressing Hashing

- Insert 200
  - 200%16 = 8
  - But slot 8 is occupied
  - Put into slot 9



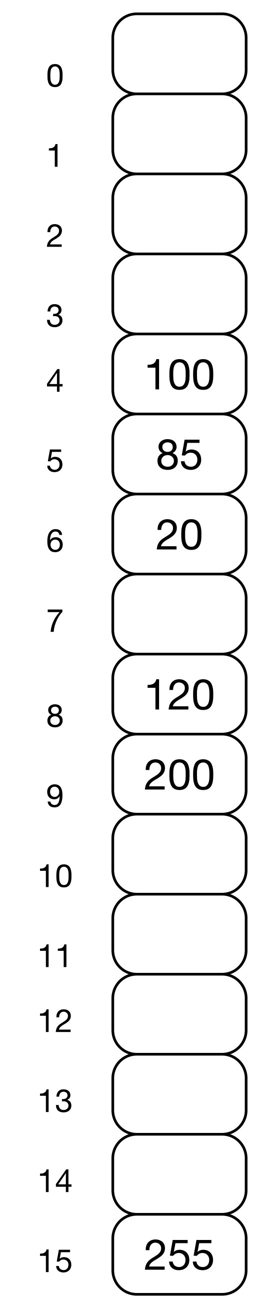
Open Addressing Hashing

- Insert 255
    - 255%16 = 15



Open Addressing Hashing

- Insert 20
  - 20%16 = 4
  - Try slot 4
  - Then slot 5
  - Then insert into slot 6



Open Addressing Hashing

- Linear probing implementation

```
class Hashtable:
    def __init__(self, slots):
        self.array = [None]*slots
        self.hash = lambda x : x%slots
```

Open Addressing Hashing

- Linear probing implementation

```
class Hashtable:
   def __repr__(self):
        retVal = ''
        for i in range(len(self.array)):
            retVal += '{}: {}\n'.format(i,  self.array[i])
        return retVal
```

Open Addressing Hashing

- Linear probing implementation

```
class Hashtable:
   def insert(self, key, value):
        slot = self.hash(key)
        while True:
            if self.array[slot]:
                slot = (slot+1).len(self.array)
            else:
                self.array[slot] = (key,value)
                return
```

Open Addressing Hashing

- Notice that the next slot can wrap around
  - Need to reset it to zero then

```
class Hashtable:
   def insert(self, key, value):
        slot = self.hash(key)
        while True:
            if self.array[slot]:
                slot = (slot+1).len(self.array)
            else:
                self.array[slot] = (key,value)
                return
```

Open Addressing Hashing

- Linear probing: Reading
  - We need to follow the same sequence of slots

```
def read(self, key):
        slot = self.hash(key)
        while True:
            if not self.array[slot]:
                return None
            else:
                if key == self.array[slot][0]:
                    return self.array[slot][1]
                else:
                    slot = (slot+1)%len(self.array)
```

Open Addressing Hashing

- This code contains an unspoken assumption:
  - There is a free slot:
    - Otherwise, we will loop forever!

```
def read(self, key):
        slot = self.hash(key)
        while True:
            if not self.array[slot]:
                return None
            else:
                if key == self.array[slot][0]:
                    return self.array[slot][1]
                else:
                    slot = (slot+1)%len(self.array)
```

Open Addressing Hashing

- E.g. use a for loop

```
def read(self, key):
        slot = self.hash(key)
        for i in range(len(self.array)):
            if not self.array[slot]:
                return None
            else:
                if key == self.array[slot][0]:
                    return self.array[slot][1]
                else:
                    slot = (slot+1)%len(self.array)
```

Open Addressing Hashing

- This type of unspoken assumption can destroy your application
    - A bug that only happens under very specific circumstances
    - Address this by
        - limiting the loop to at most m iterations

Open Addressing Hashing

- Intuitive Analysis for failed search with probing
  - We go to the slot $h(\text{key})$:
    - 1 access
  - With probability α, that slot is occupied and we need to go to the next one:
    - $1 + \alpha$ accesses
    - With probability α, that next one is occupied too, with total probability α^2:
      - $1 + \alpha + \alpha^2$ accesses

Open Addressing Hashing

- Intuitive Analysis for failed search with probing
  - In total: $1 + \alpha + \alpha^2 + \alpha^3 + \ldots$ accesses
  - $$= \frac{1}{1-\alpha}$$
- E.g.: $\alpha = 0.5$: two slots accessed
- E.g.: $\alpha = 0.9$: ten slots accessed

Open Addressing Hashing

- Expected number in a successful search:
  - $\frac{1}{\alpha}\log_e(\frac{1}{1-\alpha})$
- E.g.: $\alpha = 0.5$: 1.387 slots accessed
- E.g.: $\alpha = 0.9$: 2.559 slots accessed

Open Addressing Hashing

- Probe sequence
  - Linear probing for key c:
    - $h(c), h(c) + 1, h(c) + 2, \ldots$
  - Can lead to conveying / **primary clustering**:
    - Contiguous areas of slots
- Use a **secondary** hash function h_2
  - Should have a range co-prime to the number of slots
  - Linear probing with secondary hash function for key c
    - $h(c), h(c) + 1 \cdot h_2(c), h(c) + 2 \cdot h_2(c), h(c) + 3 \cdot h_2(c), \ldots$

Open Addressing Hashing

- Quadratic probing:
  - Use a probe sequence
    - $h(c), h(c) + 1^2, h(c) + 2^2, h(c) + 3^2, h(c) + 4^2, \ldots$
      - wrapping around 0, i.e. modulo m

Open Addressing Hashing

- In practice:
  - Linear probing can still be faster because cache loads transfer contiguous sets of memory

Hashing

- Hash schemes work extremely well:
  - If load factor can be estimated
- If size of the hash table needs to grow dynamically, things are no longer so easy
  - Extendible hashing
  - Linear hashing