EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
HUMANITIES

Department of General and Computational Linguistics

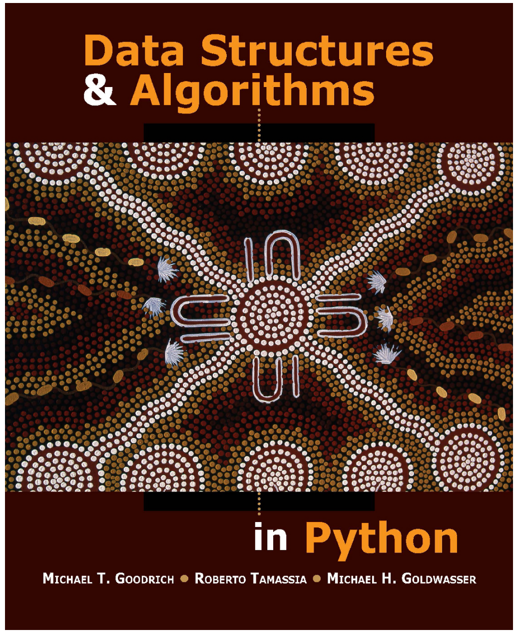
Hash Tables

Data Structures and Algorithms for CL III, WS 2019-2020

Corina Dima
corina.dima@uni-tuebingen.de

Data Structures & Algorithms in Python

MICHAEL GOODRICH
ROBERTO TAMASSIA
MICHAEL GOLDWASSER



10.1 Maps and Dictionaries

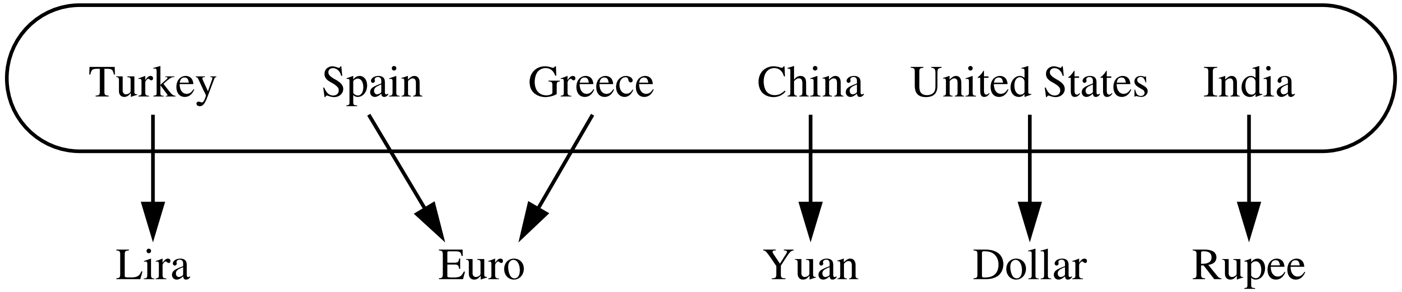
- The Map ADT

10.2 Hash Tables

- Hash Functions
- Collision-Handling Schemes
- Load Factors, Rehashing and Efficiency
- Hash Table Implementations

Maps

- map abstraction: unique keys are mapped to associated values
- maps are also known as associative arrays or dictionaries
- Python's dict class is an implementation of the map ADT



Map of countries (keys) associated with their currency (values)

- The keys are assumed to be unique, but the values are not necessarily unique
- An array-like syntax is used
  - To obtain the value associated with a key: currency['Spain']
  - To remap the key to a new value: currency['Greece'] = 'drachma'
- However, unlike in an array, indices don't have to be consecutive – and not even numeric

The Map ADT (1) – Core Functionality

M[k]	Return the value v associated with the key k in map M, if one exists; otherwise raise a KeyError; in Python, implemented with the __getitem__ method.
M[k] = v	Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k. In Python, implemented using the __setitem__ method.
del M[k]	Remove from map M the item with key equal to k; if M has no such item, raise a KeyError. In Python implemented with the __delitem__ method.
len(M)	Return the number of items in map M. In Python, implemented with the __len__ method.
iter(M)	The default iteration for a map generates a sequence of keys in the map. In Python, implemented with the __iter__ method – allows loops of the form: for k in M

The Map ADT (2)

k in M	Return True if the map contains an item with key k. In Python, implemented with the __contains__ method.
M.get(k, d=None)	Return M[k] if key k exists in the map; otherwise return default value d. This provides a way to query M[k] without the risk of a KeyError.
M.setdefault(k, d)	If key k exists in the map, return M[k]. If k does not exist, set M[k] = d and return that value.
M.pop(k, d=None)	Remove the item associated with key k from the map and return its associated value v. If key is not in the map, return default value d (or raise KeyError if d is None).
M.popitem()	Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. Raise KeyError if M is empty.
M.clear()	Remove all key-value pairs from the map.
M.keys()	Return a set-like view of all keys in M.
M.values()	Return a set-like view of all values in M.
M.items()	Return a set-like view of (k,v) tuples for all entries in M.
M.update(M2)	Assign M[k] = v for every (k,v) pair in M2.

MapBase

```
class MapBase(MutableMapping):
  """Our own abstract base class that includes a nonpublic _Item class."""

  #------------------------------ nested _Item class ------------------------------
  class _Item:
    """Lightweight composite to store key-value pairs as map items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __eq__(self, other):
      return self._key == other._key     # compare items based on their keys

    def __ne__(self, other):
      return not (self == other)         # opposite of __eq__

    def __lt__(self, other):
      return self._key < other._key      # compare items based on their keys
```

Python's MutableMapping Abstract Base Class

- Python's collections module provides two abstract base classes for working with maps: Mapping and MutableMapping
- The Mapping class contains the nonmutating behaviors supported by Python's dict class
- The MutableMapping class extends the Mapping class to include mutating behaviours
- These are abstract base classes (ABCs) – they contain methods that are declared to be abstract
- Such methods must be implemented by concrete subclasses
- However, the ABC provides concrete implementations that depend on the use of the abstract implementations
  - E.g. MutableMapping provides implementations for all the operations on the slide 5
  - But it depends on the concrete subclass to provide implementations for the core functionality (listed on slide 4)
  - the behaviors on s. 5 can be inherited by declaring MutableMapping as a parent class

Unsorted Map Implementation

```
class UnsortedTableMap(MapBase):
  """Map implementation using an unordered list."""

  def __init__(self):
    """Create an empty map."""
    self._table = [ ]                          # list of _Item's

  def __getitem__(self, k):
    """Return value associated with key k (raise KeyError if not found)."""
    for item in self._table:
      if k == item._key:
        return item._value
    raise KeyError('Key Error: ' + repr(k))

  def __setitem__(self, k, v):
    """Assign value v to key k, overwriting existing value if present."""
    for item in self._table:
      if k == item._key:                       # Found a match:
        item._value = v                        # reassign value
        return                                 # and quit
    # did not find match for key
    self._table.append(self._Item(k,v))

  def __delitem__(self, k):
    """Remove item associated with key k (raise KeyError if not found)."""
    for j in range(len(self._table)):
      if k == self._table[j]._key:             # Found a match:
        self._table.pop(j)                     # remove item
        return                                 # and quit
    raise KeyError('Key Error: ' + repr(k))

  def __len__(self):
    """Return number of items in the map."""
    return len(self._table)

  def __iter__(self):
    """Generate iteration of the map's keys."""
    for item in self._table:
      yield item._key                          # yield the KEY
```

Hash Tables

Warmup: Lookup Tables

- a map M supports the abstraction of using keys as indices using the M[k] syntax
- Consider a restricted setting in which a map with n items uses keys that are known to be integers from 0 to $N-1$, with $N \geq n$.
- We could then represent the map using what is known as a lookup table of size N

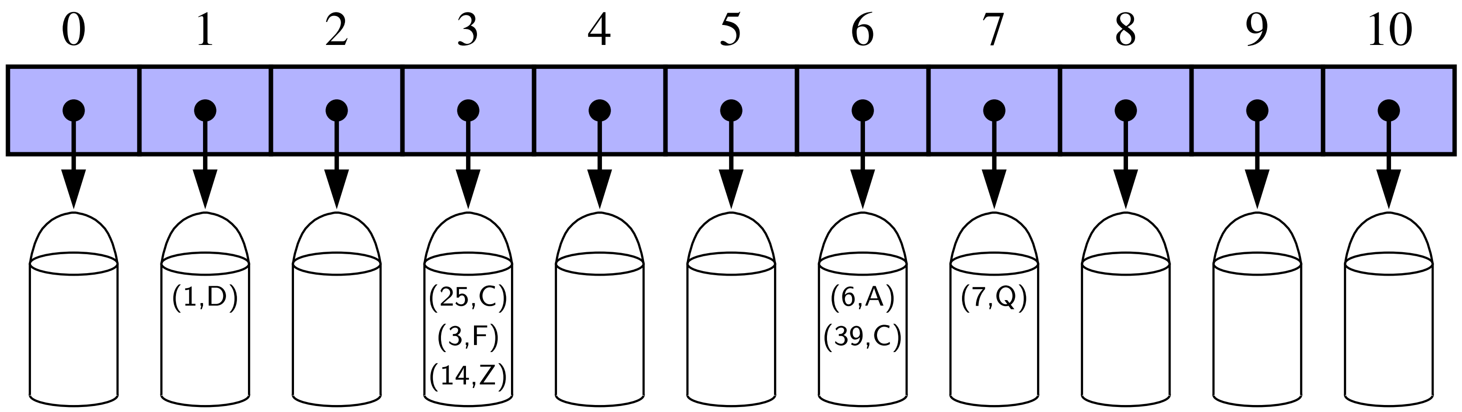
0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Lookup table with length 11 for a map containing the items (1,D), (3,Z), (6,C), (7,Q)

- However, the lookup table is not very practical
  - If $N \gg n$, the map representation uses too much space
  - The keys of the map must be integers

Hash Tables

- Instead of requiring the keys to be integers, use a hash function to map any key to a range 0 to $N-1$
- Ideally, the indices (keys) obtained via a hash function should be well (uniformly) distributed over the 0 to $N-1$ range, but in practice there might be distinct keys that get mapped to the same index
- Conceptualize the hash table as a bucket array – each bucket may manage a collection of items that are assigned the same index by the hash function

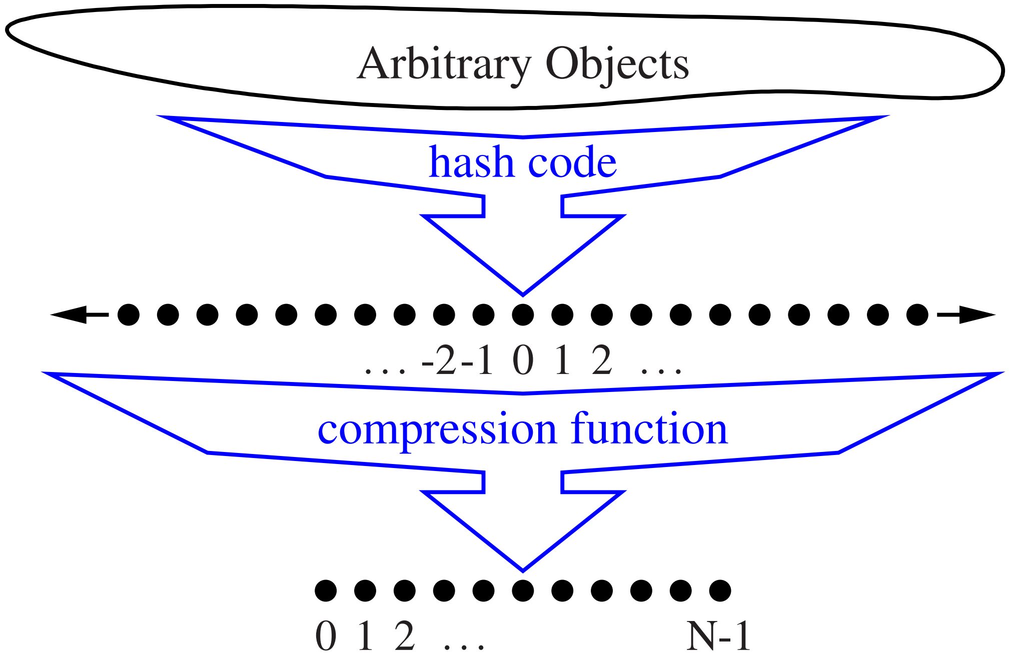


Hash Functions

- The goal of a hash function h is to map each key k to an integer in the range $[0, N-1]$, where N is the capacity of the bucket array for the hash table
- Instead of using directly the key k as an index in the array, which might not be appropriate, use the hash function value, $h(k)$, as the index
  - E.g. for the bucket array A, the item (k, v) will be stored in the bucket $A[h(k)]$
- If two or more keys have the same hash value, then two different items will be mapped to the same bucket in A – this is called a hash collision
- There are multiple strategies for dealing with hash collisions: separate chaining, open addressing
- A hash function is good if:
  - It maps the keys in the map as to sufficiently minimize collisions
  - It is fast and easy to compute

Hash Functions (cont'd)

- A hash function, $h(k)$ typically consists of two parts:
    1. A hash code that maps a key k to an integer
    2. A compression function that maps the hash code to an integer within a range of integers, $[0, N-1]$ for a bucket array
- Separating the two parts makes it possible to compute the hash code independently of the specific hash table size
- Only the compression function depends on the size of the hash table – important, especially since the underlying array can be resized



Hash Codes

- The hash code for an arbitrary key k is
  - an integer
  - doesn't have to be in the range $[0,\ N-1]$
  - may even be negative
- The set of hash codes assigned to the keys should avoid collisions as much as possible
- If the hash codes already generate collisions, there is no way for them to be avoided in the compression step
- (some) possible types of hash codes:
  - Bit representations
  - Polynomial hash codes
  - Cyclic-shift hash codes

Bit Representation as a Hash Code

- For any data type X, we can take as a hash code for X an integer interpretation of its bits
  - E.g. hash code for 803 could be 803
  - E.g. hash code for 3.14 could be based upon an interpretation of the bits of the floating-point representation as an integer
- Not applicable for types where the representation is longer than the desired hash code size
  - E.g. transform a 64-bit key to a 32-bit hash code
  - Solution 1: discard a part of the representation (rely only on the high-order or low-order bits) – might lead to many keys colliding, since part of the information is discarded
  - Solution 2: combine all the bits from the original representation into a representation – e.g. add the two 32-bit representations, ignoring overflow, or do an exclusive-or

$\sum_{i=0}^{n-1} x_i$ or $x_0 \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1}$, $\oplus$ is exclusive-or (XOR) (^ in Python)

Polynomial Hash Codes

- For character strings or other variable-length objects that can be seen as tuples of the form $(x_0, x_1, \dots, x_{n-1})$, where the order of the x_i's is significant, summation or exclusive-or hash codes are not a good solution
- E.g. a 16-bit hash code for a character string s that sums the Unicode values of the characters in s will produce collisions for common groups of strings: *stop*, *tops*, *pots* and *spot* will all have the same hash code
- A better solution is to take into consideration the positions of each x_i:

$$x_0 a^{n-1} + x_1 a^{n-2} + \dots + x_{n-2} a + x_{n-1}, \text{ for } a \neq 0, a \neq 1$$

- This is a polynomial in a that takes the components $(x_0, x_1, \dots, x_{n-1})$ of an object x as its coefficients
- can be computed in linear time using Horner's rule

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \dots + a(x_2 + a(x_1 + a\, x_0)) \dots))$$

Polynomial Hash Codes (cont'd)

- When computing the polynomial, overflows can occur – they are typically ignored
- The choice of a has an influence over the ability of the hash code to preserve some of the information content even in overflow cases
- Experimental studies suggest that 33, 37, 39 and 41 are good choices for a when working with character strings that are English words
  - E.g. when using 33, 37, 39 and 41 less then 7 collisions were produced (in each case) for the hash codes of words form a 50,000 word list

Cyclic-Shift Hash Codes

- Variant of the polynomial hash code
- Replaces multiplication by a by a cyclic shift of a partial sum by a certain number of bits
- E.g. a 5-bit cyclic shift of the 32-bit value

$$\underline{00111}101100101101010100010101000$$

is

$$101100101101010100010101000\underline{00111}$$

- The cyclic-shift operation has little in terms of meaning - but accomplishes the goal of varying the bits of the hash code
- In Python a cycling-shift of bits can be obtained using the bitwise operators $\ll$ and $\gg$ - the results must also be truncated to 32 or 64 bits.

Cyclic-Shift Hash Codes – Python implementation

```
def hash_code(s):
  mask = (1 << 32) - 1                       # limit to 32-bit integers
  h = 0
  for character in s:
    h = (h << 5 & mask) | (h >> 27)          # 5-bit cyclic shift of running sum
    h += ord(character)                      # add in value of next character
  return h
```

Cyclic-Shift Hash Codes (cont'd)

- As with the polynomial hash codes, choosing the amount by which each code should be shifted must be fine-tuned
- E.g. the collision behavior for a cyclic-shift hash code shifting from 0 to 16 bits for a list of just over 230,000 English words
- The column "Total" records the total number of words that collide with at least one another
- The "Max" column records the maximum number of words colliding at any one hash code
- shift = 0 – just sums all the characters

Shift	Collisions Total	Collisions Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

Hash Codes in Python

- The standard mechanism for computing hash codes in Python is a built-in function, hash(x), that returns an integer value that serves as a hash code for object x
- Only immutable datatypes are hashable in Python – to ensure that the hash code of a particular object remains constant during its lifetime
- int, float, str, tuple and frozenset all produce robust hash codes via the hash function
- Hash codes for character strings are based on a technique similar to polynomial hash codes which uses exclusive-or computations instead of additions
  - A total of only 8 string collide in the 230,000 strings example using Python's builtin hash function for strings
- Hashes for tuples are based on a similar technique – are based upon a combination of the hash codes of the individual elements of the tuple
- If hash(x) is called for an instance x of a mutable type, e.g. a list, a TypeError is raised

Hash Codes in Python (cont'd)

- Instances of user-defined classes are unhashable by default – calling hash() on such instances will lead to a TypeError if hash() is not overriden
- Cannot use user-defined classes as keys in a dict unless __hash__ is defined
- A function that computes the hash code can be implemented via the __hash__ method within the class
  - The returned hash code should reflect the immutable attributes of an instance
  - E.g. for a Color class that maintains three numeric red, green and blue components an implementation might be

```
def __hash__(self):
  return hash( (self._red, self._green, self._blue) )    # hash combined tuple
```

- Also, if a class defines equivalence through __eq__, then any implementation of __hash__ must be consistent, i.e. if x == y, then hash(x) == hash(y)
  - E.g. in Python 5 == 5.0, so hash(5) and hash(5.0) are the same

Compression Functions

- The hash code for a key k might not be immediately usable in a bucket array – the returned integer might be negative, or might exceed the capacity of the bucket array
- The task of the compression function:
  - map the hash code for a key k to the range $[0, N-1]$ of indices in the bucket array
- A good compression function will minimize the set of collisions for a given set of distinct hash codes
  - The division method
  - The MAD method

Compression Functions: The Division Method

- Maps an integer i to $i \bmod N$, where N is the size of the bucket array and is a fixed, positive integer
- If we choose N to be a prime number, this compression function will help "spread out" the distribution of hashed values – ideally we would want a uniform distribution
  - If N is not prime, there is a greater chance of collision due to repeating patterns
  - E.g. insert keys with hash codes 200, 205, 210, 215, 220, …, 600 into a bucket array of size 100
    - 200 mod 100 = 0, 300 mod 100 = 0, 400 mod 100 = 0, 500 mod 100 = 0, 600 mod 100 = 0
    - 205 mod 100 = 5, 305 mod 100 = 5, 405 mod 100 = 5, 505 mod 100 = 5
    - 210 mod 100 = 10, 310 mod 100 = 10, 410 mod 100 = 10, 510 mod 100 = 10
    - 215 mod 100 = 15, …
    - 220 mod 100 = 20, …

Compression Functions: The Division Method (cont'd)

- But if the bucket size is 101, there are no collisions
  - 200 mod 101 = 99, 300 mod 101 = 98, 400 mod 101 = 97, 500 mod 101 = 96, 600 mod 101 = 95
  - 205 mod 101 = 3, 305 mod 101 = 2, 405 mod 101 = 1, 505 mod 101 = 0
  - 210 mod 101 = 8, 310 mod 101 = 7, …
  - 215 mod 101 = 13

- If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is $1/N$ (uniform)
- Choosing N to be a prime number might not be enough – if there is a repeated pattern of hash codes of the form $pN + q$ for different p values, there will still be collisions

Compression Functions: The MAD Method

- The Multiply-Add-and-Divide (MAD) method maps an integer i to

$$[(ai + b) \bmod p] \bmod N$$

- Where
  - N is the size of the bucket array
  - p is a prime number larger than N
  - a and b are integers chosen at random from the interval $[0, p-1]$, with $a > 0$
- This compression function eliminates repeated patterns in the set of hash codes, making it less likely that two different keys will collide

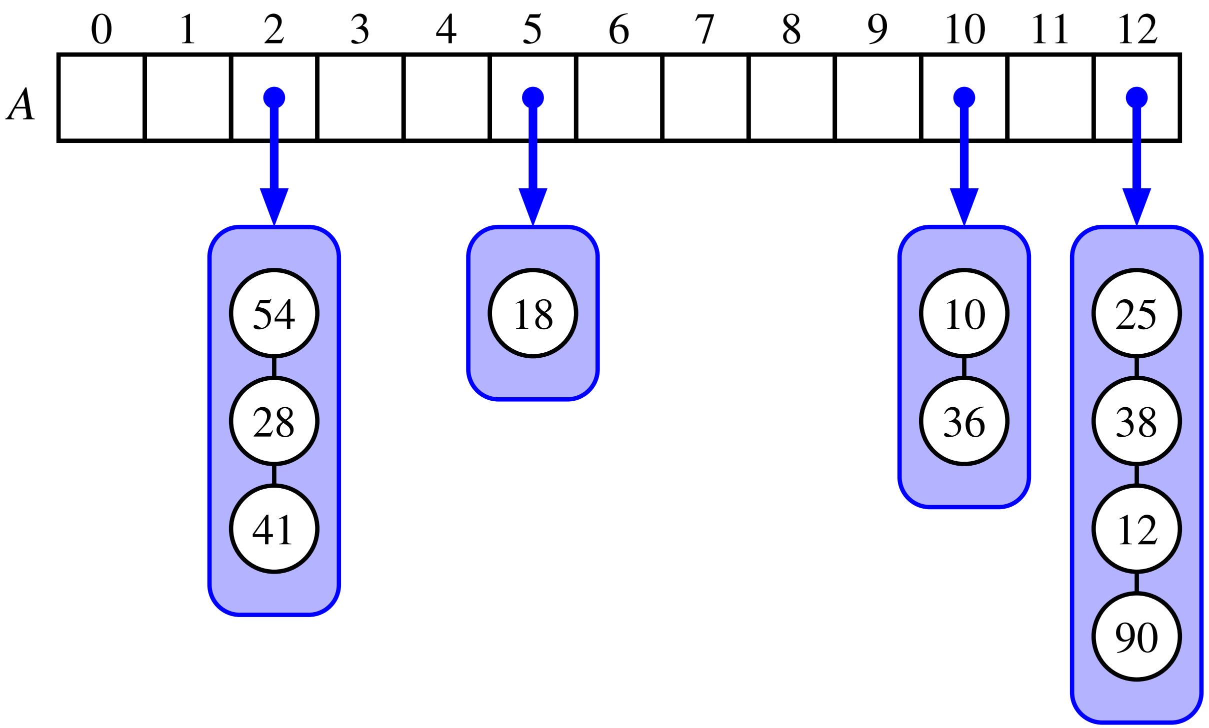
Collision-Handling Schemes

Collision-Handling Schemes

- Main idea of a hash table: take a bucket array A and a hash function h, and use them to implement a map by storing each item (k, v) in the bucket - $A[h(k)] = v$
- However, having a simple bucket array doesn't work if there are two distinct keys k_1 and k_2 for which the hash function produces the same hash code, $h(k_1) = h(k_2)$
- Such collisions prevent us from being able to add item (k_2, v_2) once (k_1, v_1) was added
- Additional care needed to deal with such collisions when inserting, searching for and deleting elements from the map

Collision Handling via Separate Chaining

- Each bucket $A[j]$ stores its own secondary container, holding all the items (k, v) such that $h(k) = j$ – e.g. use a list to implement the secondary container



Hash map of size 13, storing 10 items. Hash function is $h(k) = k \bmod 13$.

Collision Handling via Separate Chaining (cont'd)

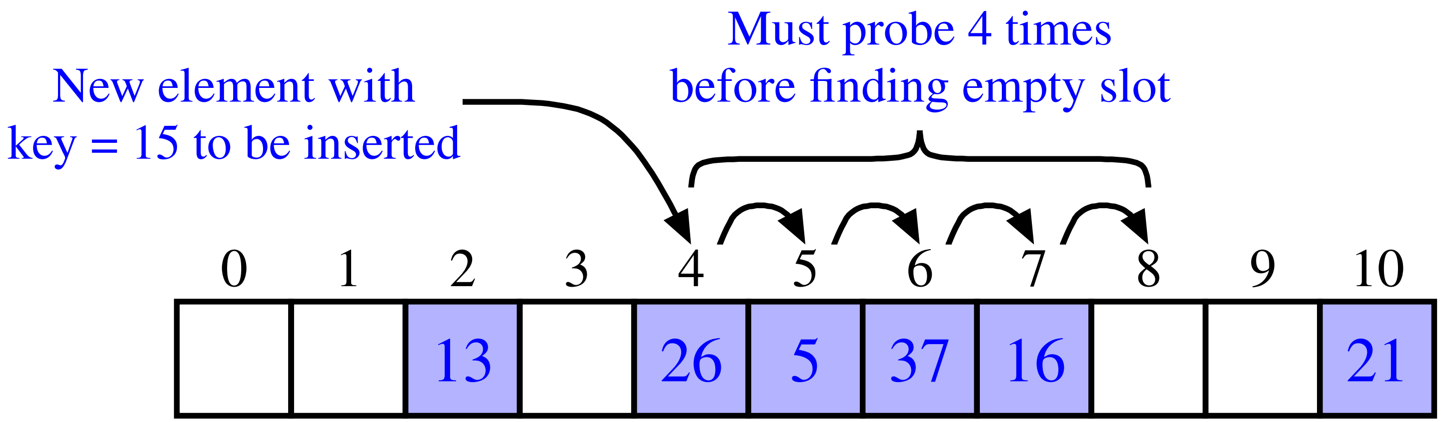
- Worst case: operations on an individual bucket take time proportional to the size of the bucket
- For a good hash function which spreading n items uniformly in a bucket array of size N, the expected bucket size is n/N
- Therefore, for a good hash function, the core map operations will run in $O(\lceil n/N \rceil)$ time
- $\lambda = n/N$ is called the load factor of the hash table
  - Should be bounded by a small constant, e.g. 1
  - Then the hash table operations run in $O(1)$ expected time

Collision Handling via Open Addressing

- The separate chaining mechanism is nice and simple, however, it does require the use of an auxiliary data structure – a list – to hold items with colliding keys
- If space is an issue (e.g. consider hand-held devices with little memory), then a set of alternative approaches can be used, which store the colliding items directly in the original bucket array
- Downside:
  - More complex algorithms for storing, retrieving and removing items from the map

Collision Handling via Open Addressing: Linear Probing

- Linear probing:
  - When we try to insert an item (k, v) into a bucket $A[j]$ that is already occupied, where $j = h(k)$, then we try next $A[(j+1)\ mod\ N]$
  - If $A[(j+1)\ mod\ N]$ is free, insert item at this position
  - Otherwise, check if $A[(j+2)\ mod\ N]$ is free, and so on, until an empty bucket is found.



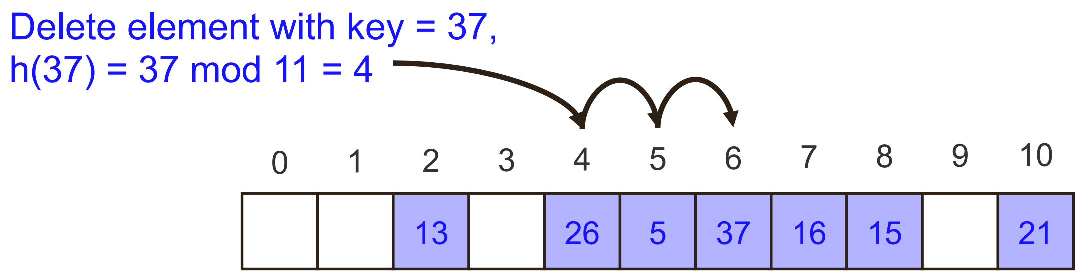
Insertion into a hash table with integer keys using linear probing, $h(k) = k\ mod\ 11$

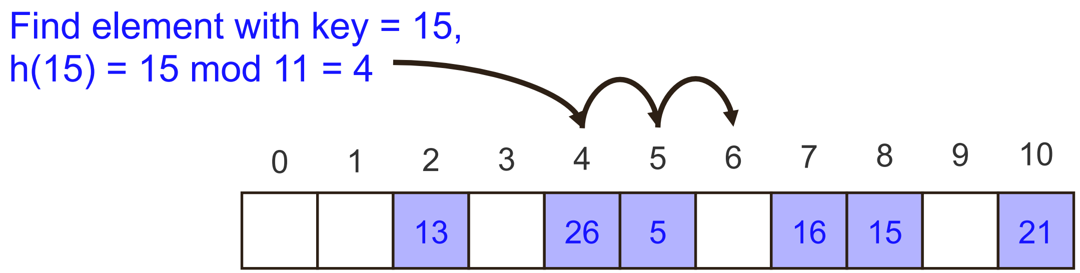
Collision Handling via Open Addressing: Linear Probing (cont'd)

- The linear probing collision strategy requires changes in implementation when searching for a particular key – when implementing:
  - __getitem__
  - __setitem__
  - __delitem__
- Called linear probing since each access of a cell of the bucket array can be seen as a "probe"
- For locating an item with key equal to k:
  - Examine consecutive slots starting from the position given by $h(k)$
    - Until we find the item with the key k
    - Or we find an empty bucket (meaning that the item with key k was not found in the hash table)

Collision Handling via Open Addressing: Linear Probing (cont'd)

- For deleting an item with key equal to k:
  - If we were to just delete any item, then subsequent searches might fail





The search stops because an empty cell was found – could not retrieve element with key 15 from the map.

Collision Handling via Open Addressing: Linear Probing (cont'd)

- For deleting an item with key equal to k:
  - Workaround: replace the deleted item with a special "available" marker object
  - The search function should be updated such that it skips such positions and continues probing until either finding the item with the given key, or an empty cell
  - When setting an item, such an "available" cell is a valid location for inserting a new item
- The use of open addressing can save space
- However, linear probing has a disadvantage, namely that it tends to cluster items of the map into contiguous runs – and these runs might even overlap
- Such runs of items considerably slow down the hash table operations – and tend to occur frequently if more than half of the cells of the hash table are occupied

Collision Handling via Open Addressing: Quadratic Probing

- Iteratively tries the buckets $A[(h(k) + f(i)) \ mod \ N]$ for $i = 0,1,2, \ldots$ where $f(i) = i^2$, until finding an empty bucket
- As with linear probing, extra care must be given to implementing the delete operation
- However, this method no longer exhibits the clustering patterns of the linear probing method
- It does create its own kind of clustering – secondary clustering – since the set of filled cells will still have a non-uniform pattern even with evenly distributed hash codes
- If N is prime and the bucket array is less than half full, then quadratic probing is guaranteed to find an empty slot
  - The guarantee is no longer valid if the hash table becomes at least half full, or N is not prime

Collision Handling via Open Addressing: Double Hashing

- Choose a secondary hash function, h'
- If h maps some key k to a bucket $A[h(k)]$ that is already occupied, then iteratively try the buckets $A[\big(h(k) + f(i)\big)\ mod\ N]$ next, for $i = 1,2,3, \ldots$ where $f(i) = i \cdot h'(k)$
- The secondary hash function is not allowed to evaluate to 0
- A common choice is $h'(k) = q - (k\ mod\ q)$, for some prime number $q < N$
- N should also be prime

Collision Handling via Open Addressing: Using a Pseudo-Random Number Generator

- Iteratively try buckets $A[(h(k) + f(i)) \; mod \; N]$ where $f(i)$ is based on a pseudo-random number generator
- The pseudo-random number generator provides a repeatable, yet somewhat arbitrary sequence of subsequent probes that depends on the bits of the original hash code
- This approach is used by Python's dict class

Load Factors, Rehashing and Efficiency

Load Factors

- The load factor $\lambda = \frac{n}{N}$, should ideally be kept below 1
- With separate chaining, if λ gets close to 1, the probability of a collision increases – which adds overhead to the hash table operations – since we need to resort to linear-time list operations for the buckets that have collisions
  - For hash tables with separate chaining, keeping $\lambda<0.9$ is a good rule of thumb
- With open addressing, when $\lambda > 0.5$ the clusters of entries in the bucket array start growing – due to the probing strategies searching might "bounce around" considerably before finding the element with a particular key for insertion, replacement or deletion
  - For hash tables with linear probing, $\lambda < 0.5$ is a good default
  - For hash tables with quadratic probing, double hashing or pseudo-random numbers, $\lambda < 2/3$ is a good option – e.g. this is what Python's `dict` implementation uses

Rehashing

- If an insertion causes the load factor to go above the optimum threshold for each case - rehashing:
  - Resize the underlying table (to regain a load factor under the optimum threshold)
  - Reinsert all objects into the new table
  - The hash code doesn't need to be recomputed, however, a new compression needs to be applied, which takes into account the size of the new underlying array
  - reshashing will generally scatter the items through the new bucket array
  - Typically, the new array is at least double the size of the previous one

Hash Table Efficiency

- If the hash function is good, the entries are expected to be uniformly distributed in the N cells of the bucket array
- To store n entries, the expected number of keys in a bucket is $O\lceil n/N \rceil$ - which is $O(1)$ if n is $O(N)$
- There are also costs for periodic rehashing – the table might need to be resized after a number of insertions and deletions - $O(1)^*$ - amortized cost for __setitem__ and __delitem__
- Worst case – map every item to the same bucket
  - Linear time performance when inserting one item for a hash table using separate chaining
  - Linear time performance when inserting one item when using any open addressing model where the secondary sequence of probes depends only on the hash code

Hash Table Efficiency (cont'd)

Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

Hash Tables – In Practice

- Hash tables are among the most efficient means for implementing a map
- Every programming language comes with efficient map implementations – Python's `dict`, Java's `HashMap`
- The hash table worst-case performance can serve as a means for a denial-of-service (DoS) attack
  - If the hash implementation is public, then an attacker could precompute a very large number of moderate-length strings that all hash to an identical 32-bit hash code
  - This makes all these hash codes collide with any of the discussed schemes – other than double hashing
  - With every insertion the system becomes slower, since more and more "hops" have to be made before a place for insertion is found

Hash Tables – In Practice (cont'd)

- In late 2011, such an attack was demonstrated by a team a researchers
- A typical web server will allow a series of key-value pairs to be embedded in the URL, using a syntax like ?key1=val1&key2=val2&key3=val3
- Such keys are usually stored directly in a map by a server, and the length and number of such parameters are limited with the presumption that the storage time in the map will be linear in term of the number of entries
- If all keys collide, storing the pairs takes quadratic time – causing the server to perform an inordinate amount of work
- In spring 2012, a security patch was distributed by the Python developers, introducing randomization into the computation of hash codes for strings – making it more difficult to reverse engineer a set of colliding strings

https://fahrplan.events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms.pdf

Hash Table Implementation

HashMapBase

```
class HashMapBase(MapBase):
  """Abstract base class for map using hash-table with MAD compression."""

  def __init__(self, cap=11, p=109345121):
    """Create an empty hash-table map."""
    self._table = cap * [ None ]
    self._n = 0                       # number of entries in the map
    self._prime = p                   # prime for MAD compression
    self._scale = 1 + randrange(p-1)  # scale from 1 to p-1 for MAD
    self._shift = randrange(p)        # shift from 0 to p-1 for MAD

  def _hash_function(self, k):
    return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)

  def __len__(self):
    return self._n

  def __getitem__(self, k):
    j = self._hash_function(k)
    return self._bucket_getitem(j, k)  # may raise KeyError

  def __setitem__(self, k, v):
    j = self._hash_function(k)
    self._bucket_setitem(j, k, v)             # subroutine maintains self._n
    if self._n > len(self._table) // 2:       # keep load factor <= 0.5
      self._resize(2 * len(self._table) - 1)  # number 2^x - 1 is often prime

  def __delitem__(self, k):
    j = self._hash_function(k)
    self._bucket_delitem(j, k)                # may raise KeyError
    self._n -= 1

  def _resize(self, c):            # resize bucket array to capacity c
    old = list(self.items())       # use iteration to record existing items
    self._table = c * [None]       # then reset table to desired capacity
    self._n = 0                    # n recomputed during subsequent adds
    for (k,v) in old:
      self[k] = v                  # reinsert old key-value pair
```

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

HashMapBase

- The bucket array is represented as a Python list, self._table
  - All entries are initialized to None
- self._n stores the number of distinct elements currently stored in the table
- If the load factor grows above 0.5 – rehash
- _hash_function is an utility for creating hashes based on Python's hash implementation and using a Multiply-Add-and-Divide (MAD) scheme
- HashMapBase does not define the way that the basic operations are performed
  - _bucket_getitem(j,k): search for item with key k, return it if found (or raise KeyError)
  - _bucket_setitem(j,k,v): modify bucket j by associating the key k with value v; must increment self._n
  - _bucket_delitem(j,k): remove item with key k from bucket j; decrement self._n after
  - __iter__: iterate though all the keys in the map

ChainHashMap

```
class ChainHashMap(HashMapBase):
  """Hash map implemented with separate chaining for collision resolution."""

  def _bucket_getitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError('Key Error: ' + repr(k))        # no match found
    return bucket[k]                                 # may raise KeyError

  def _bucket_setitem(self, j, k, v):
    if self._table[j] is None:
      self._table[j] = UnsortedTableMap( )       # bucket is new to the table
    oldsize = len(self._table[j])
    self._table[j][k] = v
    if len(self._table[j]) > oldsize:          # key was new to the table
      self._n += 1                             # increase overall map size

  def _bucket_delitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError('Key Error: ' + repr(k))        # no match found
    del bucket[k]                                    # may raise KeyError

  def __iter__(self):
    for bucket in self._table:
      if bucket is not None:                         # a nonempty slot
        for key in bucket:
          yield key
```

ProbeHashMap

```
class ProbeHashMap(HashMapBase):
  """Hash map implemented with linear probing for collision resolution."""
  _AVAIL = object( )       # sentinal marks locations of previous deletions

  def _is_available(self, j):
    """Return True if index j is available in table."""
    return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL

  def _find_slot(self, j, k):
    """Search for key k in bucket at index j.

    Return (success, index) tuple, described as follows:
    If match was found, success is True and index denotes its location.
    If no match found, success is False and index denotes first available slot.
    """
    firstAvail = None
    while True:
      if self._is_available(j):
        if firstAvail is None:
          firstAvail = j                      # mark this as first avail
        if self._table[j] is None:
          return (False, firstAvail)          # search has failed
      elif k == self._table[j]._key:
        return (True, j)                      # found a match
      j = (j + 1) % len(self._table)          # keep looking (cyclically)

  def _bucket_getitem(self, j, k):
    found, s = self._find_slot(j, k)
    if not found:
      raise KeyError('Key Error: ' + repr(k))          # no match found
    return self._table[s]._value

  def _bucket_setitem(self, j, k, v):
    found, s = self._find_slot(j, k)
    if not found:
      self._table[s] = self._Item(k,v)                 # insert new item
      self._n += 1                                     # size has increased
    else:
      self._table[s]._value = v                        # overwrite existing

  def _bucket_delitem(self, j, k):
    found, s = self._find_slot(j, k)
    if not found:
      raise KeyError('Key Error: ' + repr(k))          # no match found
    self._table[s] = ProbeHashMap._AVAIL               # mark as vacated

  def __iter__(self):
    for j in range(len(self._table)):                  # scan entire table
      if not self._is_available(j):
        yield self._table[j]._key
```

Thank you.