Maps, Hash tables, Sets

Python's dict class is arguably the most significant data structure in the language.

dictionaries are commonly known as associative arrays or maps

Maps use an array-like syntax for indexing

The Map ADT

Five most significant methods of a Map

M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError. In Python, this is implemented with the special method `__getitem__`.

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, this is implemented with the special method `__setitem__`.

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError. In Python, this is implemented with the special method `__delitem__`.

len(M): Return the number of items in map M. In Python, this is implemented with the special method `__len__`.

iter(M): The default iteration for a map generates a sequence of *keys* in the map. In Python, this is implemented with the special method `__iter__`, and it allows loops of the form, **for** k **in** M.

Other methods

k in M: Return True if the map contains an item with key k. In Python, this is implemented with the special __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 form to query M[k] without risk of a KeyError.

M.setdefault(k, d): If key k exists in the map, simply return M[k]; if key 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 k is not in the map, return default value d (or raise KeyError if parameter d is None).

Other methods

M.popitem( ):	Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError.
M.clear( ):	Remove all key-value pairs from the map.
M.keys( ):	Return a set-like view of all keys of M.
M.values( ):	Return a set-like view of all values of M.
M.items( ):	Return a set-like view of (k,v) tuples for all entries of M.
M.update(M2):	Assign M[k] = v for every (k,v) pair in map M2.
M == M2:	Return True if maps M and M2 have identical key-value associations.
M != M2:	Return True if maps M and M2 do not have identical key-value associations.

A lookup table to begin with

A lookup table with length 11 for a map containing items (1,D), (3,Z), (6,C), and (7,Q).

Limitations: keys confined to integers, poor memory utilisation

Hash function

A hash function is a special algorithm that takes an arbitrary input (data of any size) and converts it into a fixed-size output (hash value).

Key characteristics of a good hash function are:

Uniform Distribution: Ideally, the hash function should distribute the hash values uniformly across the available output range to avoid clustering.

Deterministic: The same input should always produce the same hash value (assuming the function hasn't changed).

Avalanche Effect: Small changes to the input should result in significant changes to the hash value. This helps to minimize collisions (when two different inputs produce the same hash value).

Hashing

Hashing is the process of applying a hash function to an input value to generate a hash code. It's a way to create a concise and unique (ideally) identifier for the data.

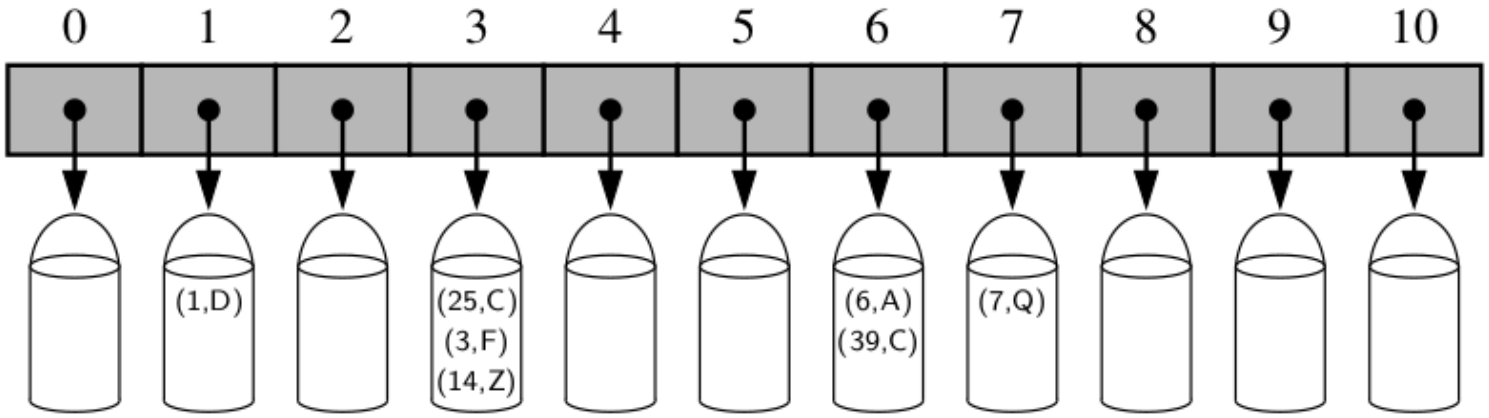
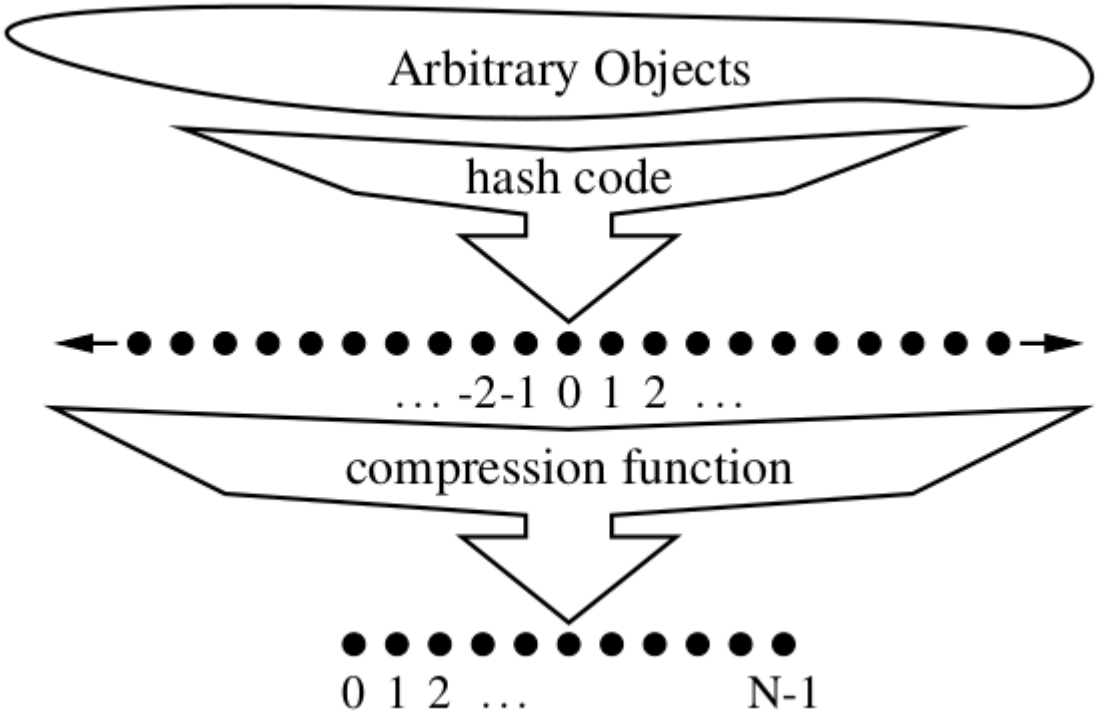


Figure 10.4: A bucket array of capacity 11 with items (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

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 a hash table.

Two components of a hash function: hash code and compression function



Hash code

Hash function performs is to take an arbitrary key k in our map and compute an integer that is called the hash code for k; this integer need not be in the range [0, N − 1], and may even be negative.

Common approaches to generate hash codes

- Bit Representation as an Integer
- Polynomial Hash Codes

$$x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}.$$

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

- Cyclic-Shift Hash Codes

Cyclic-Shift Hash Codes

```
def cyclic_hash(s):
 mask = (1 << 32) - 1
 h=0
 for ch in s:
  h = (h << 5 & mask) | (h >> 27)
  h += ord(ch)
 return h
```

Cyclic hashing of 'hello' gives hash code: 112475631

Choice of shift matters

Shift	Collisions	
	Total	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

Comparison of collision behavior for the cyclic-shift hash code as applied to a list of 230,000 English words. The "Total" column records the total number of words that collide with at least one other, and the "Max" column records the maximum number of words colliding at any one hash code. Note that with a cyclic shif of 0, this hash code reverts to the one that simply sums all the characters.

Compression Functions

1) The Division Method

A simple compression function is the division method, which maps an integer i to

i mod N

where N, the size of the bucket array, is a fixed positive integer.

if N is not prime, then it may lead to more collisions

E.g. - if we insert keys with hash codes {200, 205, 210, 215, 220, . . . , 600} into a bucket array of size 100, then each hash code will collide with at least three others. But if we use a bucket array of size 101, then there will be no collisions.

Compression Functions

2)Multiply-Add-and-Divide (or "MAD") method

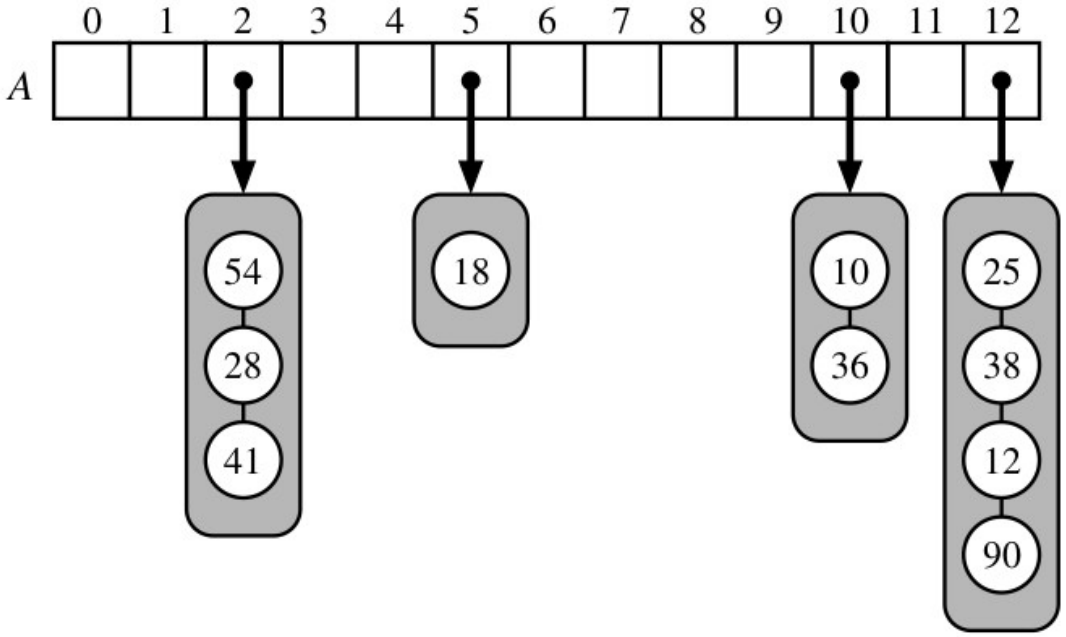
This method maps an integer i to

[(ai + b) mod p] mod N,

where N is the size of the bucket array, p is a prime number larger than N, and a and b are integers chosen at random from the interval [0, p − 1], with a > 0.

Collision-Handling Schemes

Separate Chaining



A hash table of size 13, storing 10 items with integer keys, with collisions resolved by separate chaining. The compression function is h(k) = k mod 13.For simplicity, we do not show the values associated with the keys.

Separate Chaining

Assuming we use a good hash function to index the n items of our map in a bucket array of capacity N, the expected size of a bucket is n/N.

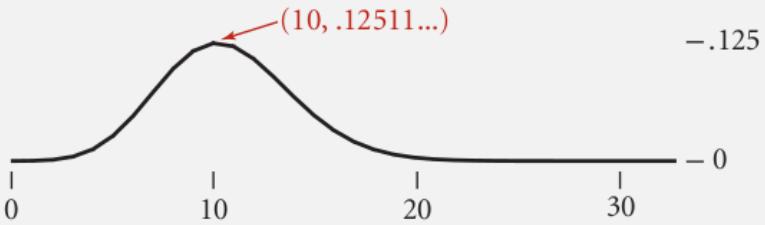
The ratio $\lambda = n/N$, called the load factor of the hash table, should be bounded by a small constant, preferably below 1.

As long as λ is O(1), the core operations on the hash table run in O(1) expected time.

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Binomial distribution ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

equals() and hashCode()

Consequence. Number of probes for search/insert is proportional to N / M.

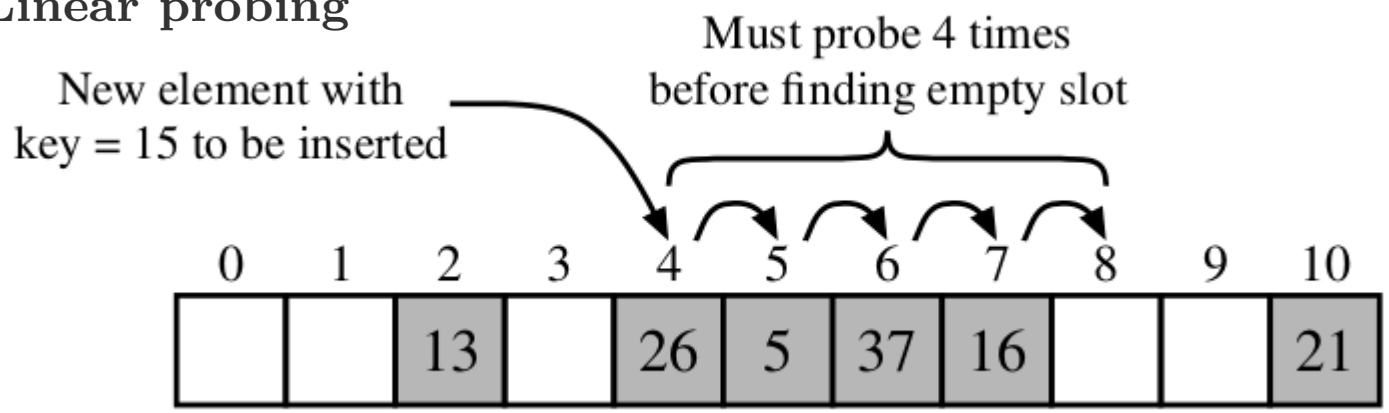
- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: $M \sim N / 4 \Rightarrow$ constant-time ops.

M times faster than sequential search

Open Addressing

Linear Probing and Its Variants

1) Linear probing



Insertion into a hash table with integer keys using linear probing. The hash function is h(k) = k mod 11. Values associated with keys are not shown.

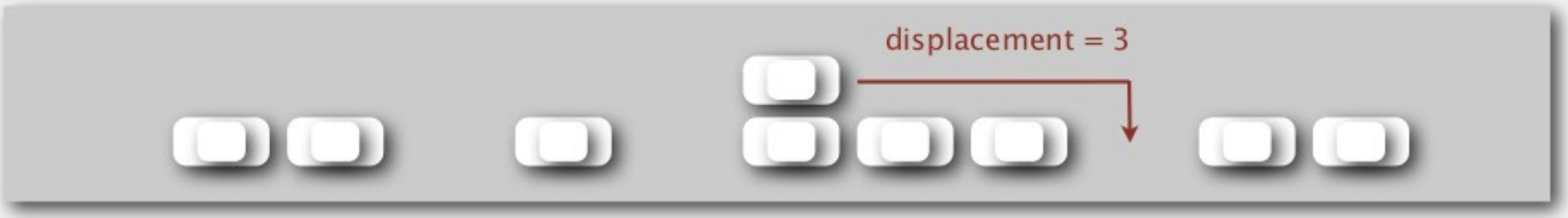
Linear probing can save space but may lead to clustering problem (particularly if more than half of the cells in the hash table are occupied). Such contiguous runs of occupied hash cells cause searches to slow down considerably

Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.

Each desires a random space i : if space i is taken, try $i+1$, $i+2$, etc.

Q. What is mean displacement of a car?



Half-full. With $M/2$ cars, mean displacement is $\sim 3/2$.

Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$.

https://algs4.cs.princeton.edu/lectures/keynote/34HashTables.pdf

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search hit | search miss / insert

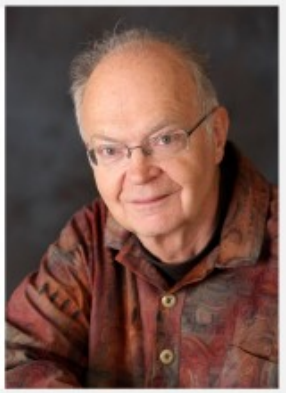
Pf.

[My first analysis of an algorithm, originally done during Summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING. D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$. ← # probes for search hit is about 3/2; # probes for search miss is about 5/2

Open Addressing

2) quadratic probing:

$$A[(h(k)+f(i)) \bmod N], \text{for } i = 0, 1, 2, \ldots, \text{where } f(i) = i^2$$

3) double hashing:

if h maps some key k to a bucket A[h(k)] that is already occupied, then we iteratively try the buckets

A[(h(k) + f (i)) mod N] next, for i = 1, 2, 3, . . .,

where f (i) = i · $h^{\#}$(k)

Open Addressing: double hashing

In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is $h^{\#}(k) = q - (k \bmod q)$, for some prime number $q < N$. Also, N should be a prime.

Efficiency of Hash Tables

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__	$O(n)$	$O(n)$	$O(n)$

Exercise problems

1. Which of the hash table collision-handling schemes could tolerate a load factor above 1 and which could not?

1. What is the worst-case time for putting n entries in an initially empty hash table, with collisions resolved by chaining? What is the best case?

1. Draw the 11-entry hash table that results from using the hash function, h(i) = (3i + 5) mod 11, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.
2. What is the result of the previous question, assuming collisions are handled by linear probing?

Exercise problems

5.Can hash table be used to implement a sorted map?

6. A hash function h defined h(key)=key mod 7, with linear probing, is used to insert the keys 44, 45, 79, 55, 91, 18, 63 into a table indexed from 0 to 6. Construct the hash table. What will be the location of key 63?

7. Given a hash table T with 25 slots that stores 2000 elements, the load factor α for T is _____.

Exercise problems

8. Consider a double hashing scheme in which the primary hash function is $h_1(k) = k \bmod 23$, and the secondary hash function is $h_2(k) = 1 + (k \bmod 19)$. Assume that the table size is 23. Then the address returned by probe 1 in the probe sequence (assume that the probe sequence begins at probe 0) for key value $k = 90$ is _____.

8. Consider a hash table of size $m = 10000$, and the hash function $h(K) = \text{floor}\,(m(KA \bmod 1))$ for $A = (\sqrt{5} - 1)/2$. The key 123456 is mapped to location _____.

Which of the following: 46, 43, 41, 48

Exercise problems

10. Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for *ii* ranging from 0 to 2020?

a. $h(i) = i^2 \bmod 10$

b. $h(i) = i^3 \bmod 10$

c. $h(i) = (11 * i^2) \bmod 10$

d. $h(i) = (12 * i) \bmod 10$

Sorted Maps ADT

Sorted maps are sorted with respect to their keys. The keys act as the ordering element, determining the position of each key-value pair within the map.

M.find_min( ):	Return the (key,value) pair with minimum key (or None, if map is empty).
M.find_max( ):	Return the (key,value) pair with maximum key (or None, if map is empty).
M.find_lt(k):	Return the (key,value) pair with the greatest key that is strictly less than k (or None, if no such item exists).
M.find_le(k):	Return the (key,value) pair with the greatest key that is less than or equal to k (or None, if no such item exists).
M.find_gt(k):	Return the (key,value) pair with the least key that is strictly greater than k (or None, if no such item exists).
M.find_ge(k):	Return the (key,value) pair with the least key that is greater than or equal to k (or None, if no such item).

Sorted Maps ADT

M.find_range(start, stop): Iterate all (key,value) pairs with start <= key < stop. If start is None, iteration begins with minimum key; if stop is None, iteration concludes with maximum key.

iter(M): Iterate all keys of the map according to their natural order, from smallest to largest.

reversed(M): Iterate all keys of the map in reverse order; in Python, this is implemented with the __reversed__ method.

Efficiency of Sorted Map using Python lists

Operation	Running Time
len(M)	$O(1)$
k in M	$O(\log n)$
M[k] = v	$O(n)$ worst case; $O(\log n)$ if existing k
del M[k]	$O(n)$ worst case
M.find_min( ), M.find_max( )	$O(1)$
M.find_lt(k), M.find_gt(k) M.find_le(k), M.find_ge(k)	$O(\log n)$
M.find_range(start, stop)	$O(s + \log n)$ where s items are reported
iter(M), reversed(M)	$O(n)$

Performance of a sorted map, as implemented with SortedTableMap which uses list. We use n to denote the number of items in the map at the time the operation is performed. The space requirement is O(n).

Sets, Multisets, and Multimaps

- A **set** is an unordered collection of elements, without duplicates, that typically supports efficient membership tests.
- A **multiset** (also known as a bag) is a set-like container that allows duplicates.
- A **multimap** is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values.

Sets

Python provides the built-in classes frozenset and set for immutable and mutable sets, respectively

abstract base class **collections.Set** corresponds to concrete frozenset class

abstract base class **collections.MutableSet** is akin to the concrete set class

The five most fundamental behaviors for a set S:

S.add(e): Add element e to the set. This has no effect if the set already contains e.

S.discard(e): Remove element e from the set, if present. This has no effect if the set does not contain e.

e in S: Return True if the set contains element e. In Python, this is implemented with the special `__contains__` method.

len(S): Return the number of elements in set S. In Python, this is implemented with the special method `__len__`.

iter(S): Generate an iteration of all elements of the set. In Python, this is implemented with the special method `__iter__`.

Methods related to remove

S.remove(e): Remove element e from the set. If the set does not contain e, raise a KeyError.

S.pop( ): Remove and return an arbitrary element from the set. If the set is empty, raise a KeyError.

S.clear( ): Remove all elements from the set.

Boolean comparisons

The next group of behaviors perform Boolean comparisons between two sets.

S == T:	Return True if sets S and T have identical contents.
S != T:	Return True if sets S and T are not equivalent.
S <= T:	Return True if set S is a subset of set T.
S < T:	Return True if set S is a *proper* subset of set T.
S >= T:	Return True if set S is a superset of set T.
S > T:	Return True if set S is a *proper* superset of set T.
S.isdisjoint(T):	Return True if sets S and T have no common elements.

S | T:	Return a new set representing the union of sets S and T.
S |= T:	Update set S to be the union of S and set T.
S & T:	Return a new set representing the intersection of sets S and T.
S &= T:	Update set S to be the intersection of S and set T.
S ^ T:	Return a new set representing the symmetric difference of sets S and T, that is, a set of elements that are in precisely one of S or T.
S ^= T:	Update set S to become the symmetric difference of itself and set T.
S − T:	Return a new set containing elements in S but not T.
S −= T:	Update set S to remove all common elements with set T.

Tests if a set is a proper subset of another

```
def __lt__(self, other):       # supports syntax S < T
  """Return true if this set is a proper subset of other."""
  if len(self) >= len(other):
    return False               # proper subset must have strictly smaller size
  for e in self:
    if e not in other:
      return False             # not a subset since element missing from other
  return True                  # success; all conditions are met
```

Union of two sets

```
def __or__(self, other):                    # supports syntax S | T
    """Return a new set that is the union of two existing sets."""
    result = type(self)( )                  # create new instance of concrete class
    for e in self:
        result.add(e)
    for e in other:
        result.add(e)
    return result
```

In-place union of one set with another

```
def __ior__(self, other):          # supports syntax S |= T
    """Modify this set to be the union of itself an another set."""
    for e in other:
        self.add(e)
    return self                    # technical requirement of in-place operator
```

References

Data Structures and Algorithms in Python
Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Introduction to Algorithms
Leiserson,Stein,Rivest,Cormen

Algorithms, 4th Edition
Robert Sedgewick and Kevin Wayne

Few Images from the internet