Dictionaries and Sets

- dict
- set
- frozenset
- set/dict comprehensions

Dictionaries (type dict)

$$\{\, key_1 : \; value_1 , \; \ldots , \; key_k : \; value_k \,\}$$

key	value
'a'	7
'foo'	'42nd'
5	29
'5'	44
5.5	False
False	True
(3, 4)	'abc'

distinct keys,
i.e. not "=="

- Stores a mutable set of (key, value) pairs, denoted *items*, with distinct keys, i.e. *maps* keys to values
- Constructing empty dictionary: `dict()` or `{}`
- $dict\,[\,key\,]$ lookup for key in dictionary, and returns associated value. Key must be present, otherwise a KeyError is raised
- $dict\,[\,key\,] = value$ assigns value to *key*, overriding exising value if present

https://docs.python.org/3/tutorial/datastructures.html#dictionaries

Dictionaries (type dict)

Python shell

```
> d = {'a': 42, 'b': 57}
> d
| {'a': 42, 'b': 57}

> d.keys()
| dict_keys(['a', 'b'])

> list(d.keys())
| ['a', 'b']

> d.items()
| dict_items([('a', 42), ('b', 57)])

> list(d.items())
| [('a', 42), ('b', 57)]
```

```
> for key in d:
      print(key)
| a
| b

> for key, val in d.items():
      print('Key', key, 'has value', val)
| Key a has value 42
| Key b has value 57

> {5: 'a', 5.0: 'b'}
| {5: 'b'}
```

Python shell

```
> surname = dict(zip(['Donald', 'Mickey', 'Scrooge'], ['Duck', 'Mouse', 'McDuck']))
> surname['Mickey']
| 'Mouse'
```

Dictionaries (type dict)

Python shell

```
> gradings = [('A', 7), ('B', 4), ('A', 12), ('C', 10), ('A', 7)]
> grades = {}  # empty dictionary
> for student, grade in gradings:
      if student not in grades:  # is key in dictionary
          grades[student] = []
      grades[student].append(grade)
> grades
| {'A': [7, 12, 7], 'B': [4], 'C': [10]}
> print(grades['A'])
| [7, 12, 7]
> print(grades['E'])  # can only lookup keys in dictionary
| KeyError: 'E'
> print(grades.get('E'))  # .get returns None if key not in dictionary
| None
> print(grades.get('E', []))  # change default return value
| []
> print(grades.get('A', []))
| [7, 12, 7]
```

Dictionary initialization

Python shell

```
> d1 = {'A': 7, 'B': 42}
> d2 = dict([('A', 7), ('B', 42)])  # list of (key, value) pairs
> d3 = dict(A=7, B=42)              # keyword arguments to constructor
> d1 == d2 == d3
| True
```

- **Note**: *keyword initialization* only works if keys are strings which are valid keyword arguments to a function – but saves writing a lot of quotes

| and |=

new in Python 3.9
PEP 584

Python shell

```
> d1 = dict(A=1, B=2)
> d2 = dict(B=3, C=4)
> d1 | d2                           # merge dictionaries
| {'A': 1, 'B': 3, 'C': 4}          # rightmost value for 'B' wins
> {**d1, **d2, 'D': 5}              # ** inserts dictionary content
| {'A': 1, 'B': 3, 'C': 4, 'D': 5}  # rightmost value for 'B' wins
> d1 |= d2                          # same as d1.update(d2)
> d1
| {'A': 1, 'B': 3, 'C': 4}
```

Dictionary operation	Description
len(d)	Items in dictionary
d[key]	Lookup key
d[key] = value	Update value of key
del d[key]	Delete an existing key
key in d	Key membership
key not in d	Key non-membership
clear()	Remove all items
copy()	Shallow copy
get(key), get(key, default)	d[key] if key in dictionary, otherwise `None` or default
items()	*View* of the dictionaries items
keys()	*View* of the dictionaries keys
values()	*View* of the dictionaries values
pop(key)	Remove key and return previous value
popitem()	Remove and return an arbitrary item
update()	Update key/value pairs from another dictionary

https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

Tuples as dictionary keys

- A tuple can be used as a dictionary key, and parenthesis can be omitted

Python shell -

```
> d = {('a', 1): 7, ('b', 2): 42}
> d[('b', 2)]
| 42
> d['b', 2]   # same as above, parenthesis omitted
| 42
> T = [[None, None], [42, None]]  # 2D table as lists-of-lists
> T[1][0]
| 42
> T[1, 0]   # wrong, T is a list (of lists)
| TypeError: list indices must be integers or slices, not tuple
> T = {(1, 0): 42}  # 2D table as dictionary
> T[1, 0]   # dictionary lookup with tuple (1, 0) as key
| 42
> T[1][0]   # wrong, T has only one key = the tuple (1, 0)
| KeyError: 1
```

Order returned by `list(d.keys())` ?

The Python Standard Library
Mapping Types — dict

"Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end." (since Python 3.7)

docs.python.org/3/library/stdtypes.html

Python shell

```
> d = {'d': 1, 'c': 2, 'b': 3, 'a': 4}
> d['x'] = 5    # new key at end
> d['c'] = 6    # overwrite value
> del d['b']    # remove key 'b'
> d['b'] = 7    # reinsert key 'b' at end
> d
| {'d': 1, 'c': 6, 'a': 4, 'x': 5, 'b': 7}
```

Raymond Hettinger @ Twitter

See also Raymond's talk @ PyCon 2017

Modern Python Dictionaries

A confluence of a dozen great ideas

Dictionary comprehension

- Similarly to creating a list using list comprehension, one can create a set of key-value pairs:

{*key* : *value* `for` *variable* `in` *list*}

Python shell

```
> names = ['Mickey', 'Donald', 'Scrooge']
> list(enumerate(names, start=1))
| [(1, 'Mickey'), (2, 'Donald'), (3, 'Scrooge')]
> dict(enumerate(names, start=1))   # construct dict from pairs
| {1: 'Mickey', 2: 'Donald', 3: 'Scrooge'}
> {name: idx for idx, name in enumerate(names, start=1)}
| {'Mickey': 1, 'Donald': 2, 'Scrooge': 3}
```

Sets (set and frozenset)

$$\{\, value_1,\ \ldots,\ value_k \,\}$$

- Values of type `set` represent mutable sets, where "==" elements only appear once
- Do **not** support: indexing, slicing
- `frozenset` is an immutable version of `set`

Python shell

```
> S = {2, 5, 'a', 'c'}
> T = {3, 4, 5, 'c'}
> S | T
| {2, 3, 4, 5, 'a', 'c'}
> S & T
| {5, 'c'}
```

```
> S ^ T
| {2, 3, 4, 'a'}
> S - T
| {2, 'a'}
> {4, 5, 5.0, 5.1}
| {4, 5, 5.1}
```

Operation	Description
S | T	Set union
S & T	Set intersection
S – T	Set difference
S ^ T	Symmetric difference
set()	Empty set ({} = empty dictionary ⚠)
set(L)	Create set from sequence
x in S	Membership
x not in S	Non-membership
S.isdisjoint(T)	Disjoint sets
S <= T	Subset
S < T	Proper subset
S >= T	Superset
S > T	Proper superset
len(S)	Size of S
S.add(x)	Add x to S (not frozenset)

https://docs.python.org/3/tutorial/datastructures.html#sets
https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

Question – What value has the expression ?

```
sorted( { 5, 5.5, 5.0, '5' } )
```

a) `{'5', 5, 5.0, 5.5}`

b) `{5, 5.5}`

c) `['5', 5.0, 5.5]`

d) `['5', 5, 5.5]`

e) TypeError

f) Don’t know

Sets of (frozen) sets

- Sets are mutable, i.e. cannot be used as dictionary keys or elements in sets
- Frozen sets can

Python shell

```
> S = {{'a'}, {'a', 'b'}, {'a', 'c'}}
| TypeError: unhashable type: 'set'
> S = {frozenset({'a'}), frozenset({'a', 'b'}), frozenset({'a', 'c'})}
> frozenset({'a', 'b'}) in S
| True
> {'a', 'b'} in S  # automatically converts unhashable set to frozenset
| True
> {'a', 'b'} == frozenset(['a', 'b'])  # frozenset from list
| True
> D = {frozenset({'a', 'b'}): 42}  # dictionary
> frozenset({'a', 'b'}) in D
| True
> {'a', 'b'} in D  # dictionaries are not that friendly as sets
| TypeError: unhashable type: 'set'
```

Set comprehension

- Similarly to creating a list using list comprehension, one can create a set of values (also using nested for- and if-statements):

  `{`*value* `for` *variable* `in` *list*`}`

- A value occurring multiple times as *value* will only be included once

primes_set.py

```
n = 101
not_primes = {m for f in range(2, n) for m in range(2 * f, n, f)}
primes = set(range(2, n)) - not_primes
```

Python shell

```
> L = ['a', 'b', 'c']
> {(x,(y,z)) for x in L for y in L for z in L if x != y and y != z and z != x}
| {('a',('b','c')),('a',('c','b')),('b',('a','c')),...,('c',('b','a'))}
> L = {'a', 'b', 'c'}
> {(x,(y,z)) for x in L for y in L - {x} for z in L - {x, y}}
| {('c',('b','a')),('c',('a','b')),('a',('c','b')),...,('b',('a','c'))}
```

Hash, equality, and immutability

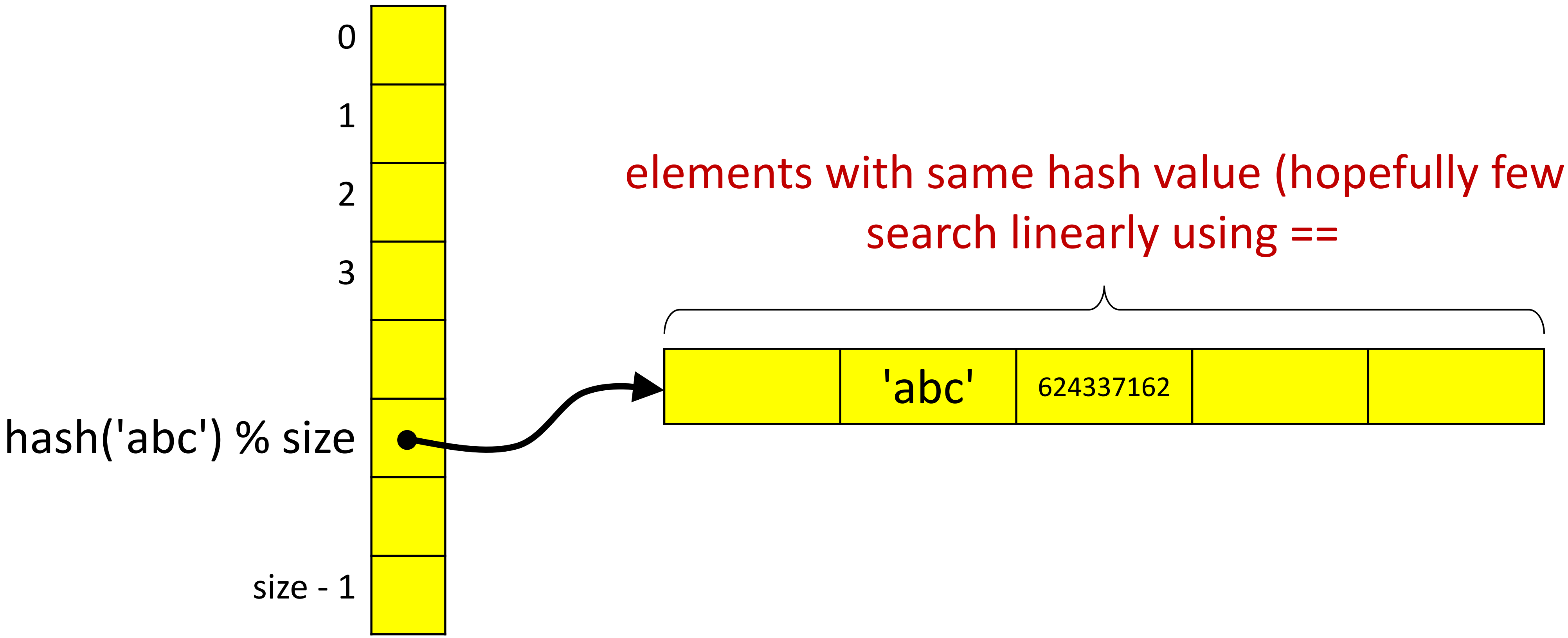
- Keys for dictionaries and sets must be *hashable*, i.e. have a `__hash__()` method returning an integer that does not change over their lifetime and an `__eq__()` method to check for equality with "=="

  `'abc'.__hash__()` could e.g. return `624337162`

  `(624337162).__hash__()` would also return `624337162`

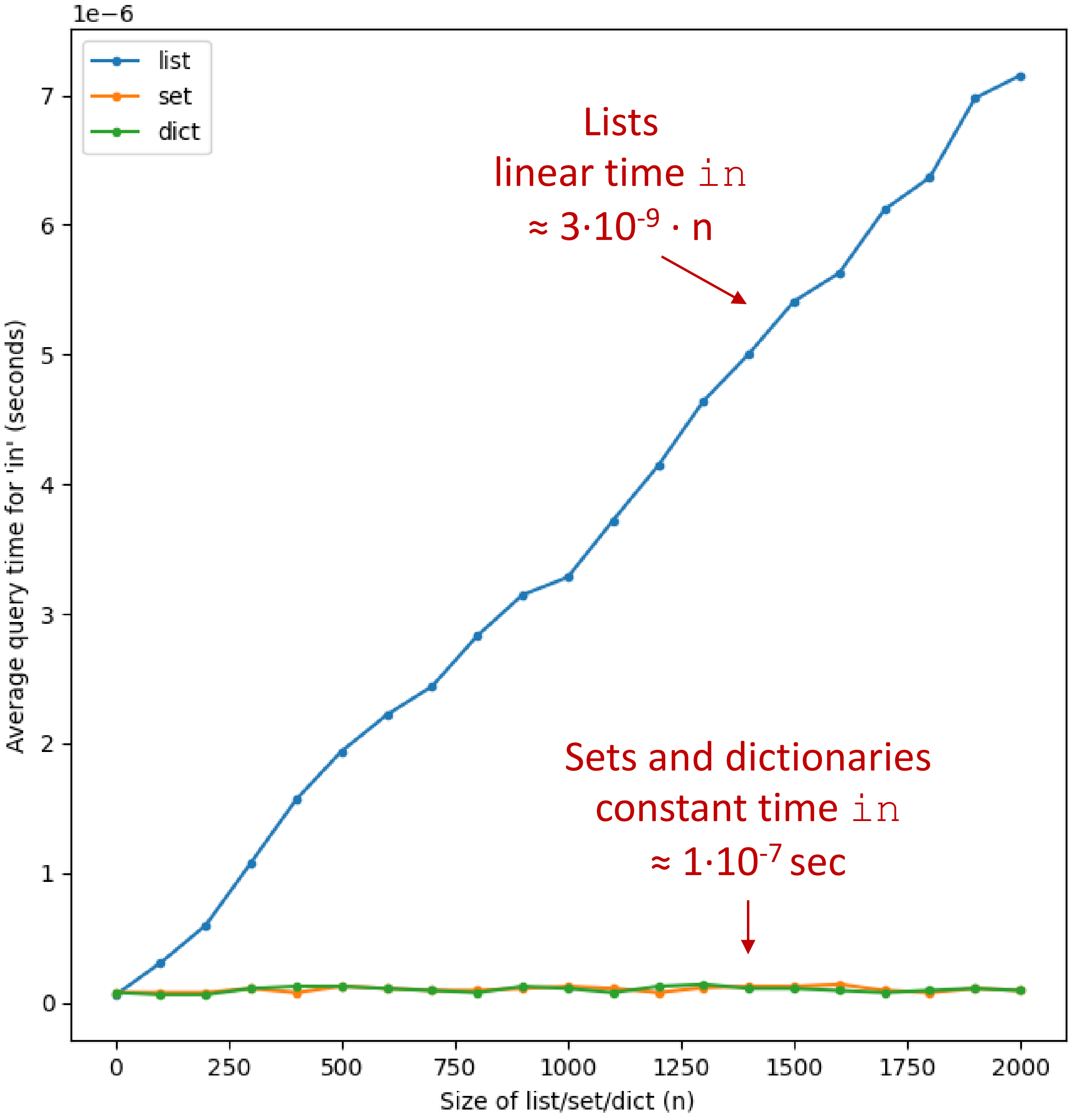
- All built-in immutable types are hashable.
  In particular tuples of immutable values are hashable.
  I.e. nested tuples like `((('a'),'b'),('c',('d','e')))` representing trees can be used as dictionary keys or stored in a set

Sketch of internal set implementation



Raymond Hettinger,
Modern Python Dictionaries
- A confluence of a dozen great ideas

Membership queries `in`



in.py

```
from random import shuffle, choices
from time import time
import matplotlib.pyplot as plt

time_L = []
time_S = []
time_D = []

ns = [1, *range(100, 2001, 100)]

for n in ns:
    print(f'{n = }')
    L = list(range(n))
    shuffle(L)
    S = set(L)
    D = {value: 42 for value in L}

    queries = choices(L, k=1_000_000)
    for X, times in [(L, time_L), (S, time_S), (D, time_D)]:
        count = 0
        start = time()
        for q in queries:
            if q in X:
                count +=1
        end = time()
        times.append((end - start) / len(queries))

plt.plot(ns, time_L, '.-', label='list')
plt.plot(ns, time_S, '.-', label='set')
plt.plot(ns, time_D, '.-', label='dict')
plt.xlabel('Size of list/set/dict (n)')
plt.ylabel("Average query time for 'in' (seconds)")
plt.legend()
plt.show()
```

⚠ Avoid using **`in`** repeatedly on long lists

Module `collections` (container datatypes)

- Python builtin containers for data: `list`, `tuple`, `dict`, and `set`.
- The module **`collections`** provides further alternatives
(but these are not part of the language like the builtin containers)

`deque`	double ended queue
`namedtuple`	tuples allowing access to fields by name
`Counter`	special dictionary to count occurrences of elements
…	

https://docs.python.org/3/library/collections.html

deque – double ended queues

- Extends lists with efficient updates at the front
- Inserting at the front of a standard Python list takes linear time in the size of the list – very slow for long lists

Python shell

```
> L = list()
> L.append(1)
> L.append(2)
> L.insert(0, 0)   # insert at the front
> L.insert(0, -1)  # slow for long lists
> L.insert(0, -2)
> L
| [-2, -1, 0, 1, 2]
```

```
> from collections import deque
> d = deque()   # create empty deque
> d.append(1)
> d.append(2)
> d.appendleft(0)   # efficient
> d.appendleft(-1)
> d.appendleft(-2)
> d
| deque([-2, -1, 0, 1, 2])
> for e in d: print(e, end=', ')
| -2, -1, 0, 1, 2,
```

`namedtuple` – tuples with field names

Since Python 3.7 `dataclass` better choice

- Compromise between `tuple` and `dict`, can increase code readability

Python shell

```
> person = ('Donald Duck', 1934, '3 feet')  # as tuple
> person[1]  # not clear what is accessed
| 1934
> person = {'name': 'Donald Duck', 'appeared': 1934, 'height': '3 feet'}  # as dict
> person['appeared']  # more clear what is accessed, but ['...'] overhead
| 1934
> from collections import namedtuple
> Person = namedtuple('Person', ['name', 'appeared', 'height'])  # create new type
> person = Person('Donald Duck', 1934, '3 feet')  # as namedtuple
> person
| Person(name='Donald Duck', appeared=1934, height='3 feet')
> person.appeared  # short and clear
| 1934
> person[1]  # still possible
| 1934
```

Counter – dictionaries for counting

Python shell

```
> from collections import Counter
> fq = Counter('abracadabra')  # create new counter from a sequence
> fq
| Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})  # frequencies of the letters
> fq['a']
| 5
> fq.most_common(3)
| [('a', 5), ('b', 2), ('r', 2)]
> fq['x'] += 5  # increase count of 'x', also valid if 'x' not in Counter yet
> Counter('aaabbbcc') - Counter('aabdd')  # counters can be subtracted
| Counter({'b': 2, 'c': 2, 'a': 1})
> Counter([1, 2, 1, 3, 4, 5]) + Counter([3, 3, 3])  # counters can be added
| Counter({3: 4, 1: 2, 2: 1, 4: 1, 5: 1})

> T = 'AfD adsf dsa f dsaf daf dsaf DSA fda f SA dsa f dsa fdsa f dsAf sAf f dsaf'
> Counter(T.lower().split()).most_common(3)
| [('f', 5), ('dsa', 4), ('dsaf', 4)]
```

Handin 3 & 4 – Triplet distance (Dobson, 1975)

```
((('A','F'),'B'),('D',('C','E')))

              (a)

            /\
           /  \
          /    \
         /      \
        /        \
       /          \
      /\          /\
     /  \        /  \
    /    \      /    \
   /\    'B'  'D'    /\
  /  \              /  \
'A'  'F'          'C'  'E'
```

How similar are the trees ?

```
(((('D','A'),'B'),'F'),('C','E'))

                (b)

                /\
               /  \
              /    \
             /      \
            /        \
           /          \
          /\          /\
         /  \        /  \
        /    \     'C'  'E'
       /      \
      /\      'F'
     /  \
    /    \
   /\    'B'
  /  \
'D'  'A'
```

arxiv.org/abs/1706.10284

Handin 3 & 4 – Triplet distance (Dobson, 1975)

Consider all $\binom{n}{3}$ subsets of size three, and count how many do not have identical substructure (topology) in the two trees.

```
     /\
    /  \
   /    \
'D'     /\
       /  \
     'A'  'F'

('D',('A','F'))
```

canonical ordering

```
              (a)

     anchor *
           / \
          /   \
         /     \
        /       \
       /         \
      /           \
     /\           /\
    /  \         /  \
   /    \       /    \
  /\    'B'  'D'     /\
 /  \               /  \
'A'  'F'          'C'  'E'
```

{'A','D','F'}

```
              (b)

               /\
              /  \
             /    \
            /      \
           /        \
          /          \
  anchor *           /\
        / \         /  \
       /   \      'C'  'E'
      /     \
     /\     'F'
    /  \
   /    \
  /\    'B'
 /  \
'D'  'A'
```

```
     /\
    /  \
   /    \
'F'     /\
       /  \
     'A'  'D'

('F',('A','D'))
```

arxiv.org/abs/1706.10284