

Alte colecții

Hash code – Funcții de dispersie

Funcții de dispersie

- ▶ Obiect *hash-uibil* = care are cod **hash** asociat, returnat de metoda `__hash__()` (= cod de dispersie, valoare hash)

= un număr întreg cu proprietățile

Funcții de dispersie

- ▶ Obiect *hash-uibil* = care are cod **hash** asociat, returnat de metoda `__hash__()` (= cod de dispersie, valoare hash)

= un număr întreg cu proprietățile

- nu se poate schimba dacă obiectul nu se modifică (de obicei imutabil)
- două obiecte **egale au același hash code**
- este recomandabil ca două obiecte diferite să aibă valori hash diferite (compatibil cu `__eq__()`)

Funcții de dispersie

- ▶ Obiect *hash-uibil* = care are **hash code** asociat, returnat de metoda `__hash__()` (= cod de dispersie, valoare hash)

= un număr întreg cu proprietățile

- nu se poate schimba dacă obiectul nu se modifică (de obicei imutabil)
- două obiecte **egale au același hash code**
- este recomandabil ca două obiecte diferite să aibă valori hash diferite (compatibil cu `__eq__()`)

Funcții de dispersie

Obiect



Număr întreg

dimensiune variabilă

plajă limitată de valori



apar coliziuni

Funcții de dispersie

```
t = (1,2)
```

```
print(hash(t)) #print(t.__hash__())
```

```
t = (1,[2,3])
```

```
print(hash(t)) # print(t.__hash__())
```

```
#TypeError: unhashable type: 'list'
```

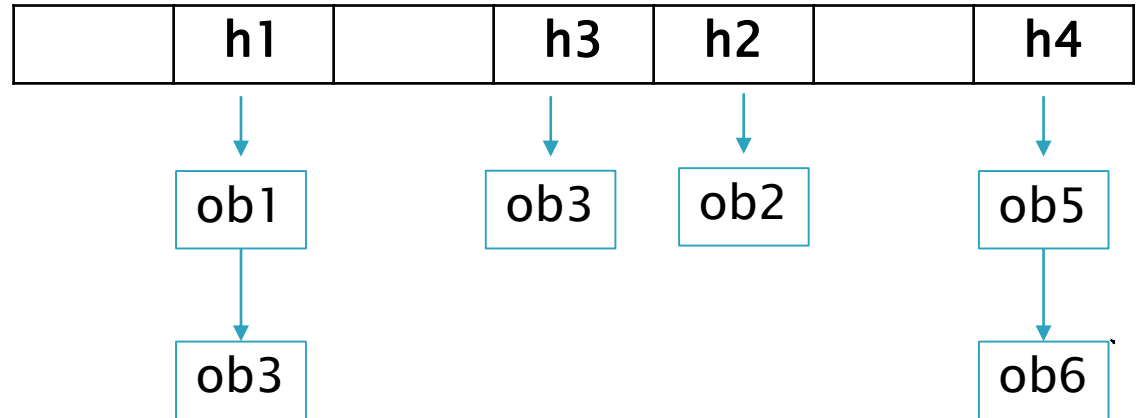


Funcții de dispersie

- ▶ Funcția hash (de dispersie) trebuie aleasă astfel încât să se **minimizeze** numărul coliziunilor (obiecte diferite care produc aceleași hash-uri).
- ▶ Obiectele hash-uibile pot fi folosite ca chei pentru structuri de date indexate după chei (se va folosi pentru indexare codul lor hash)

Tabele de dispersie

Obiect	Hash code
ob1	h1
ob2	h2
ob3	h1
ob4	h3
ob5	h4
ob6	h4



Tabele de dispersie

- ▶ structură de date pentru căutare eficientă după chei hash-uibile
- ▶ căutare – în **medie** timpul $O(1)$ (defavorabil $O(n)$)

Muṭimi

Mulțimi

Clasa set

- ▶ Elemente unice (mulțime)
- ▶ **Nu sunt indexate** (nu `s[0]`)
- ▶ Neordonată, **nu păstrează ordinea de inserare a elementelor**
- ▶ **mutabilă**
- ▶ alcătuită din elemente “*imutabile*”, de fapt *hash-uibile*

Mulțimi

Creare

```
s = {7, 5, 13}
```

```
s = {1, 2, 3, 2, 1}
```

```
s = {} #NU, este dictionar
```

```
s = {(1,2), (2,1), (3,1)} #OK
```

```
s = {[1,2], 3} #NU
```

```
#TypeError: unhashable type: 'list'
```

Muṭimi

Creare

- ▶ `set([iterabil])`
 - `s = set() #multimea vida`
 - `s = set("multime")`
 - `s = set(["multime"])`
 - `s = set(("multime"))`

Mulțimi

Creare

```
ls = [2, 3, 1, 3, 2, 6]
```

```
s = set(ls) #elementele distincte
```

```
cuv = "aceeasi"
```

```
s = set(cuv)
```



Mulțimi

Creare

- `set()` - mulțimea vida
- **!!! `s = {}` nu este multimea vida**

Multi

Creare

- Comprehensiune

```
s = {x for x in range(1,5)}
```

Mulțimi

Creare

- Comprehensiune

```
s = {x for x in range(1,5)}
```

```
ls = [2, 3, 1, 3, -2, 6, 2]
```

```
s = {x for x in ls if x>0}
```

```
#elementele distincte positive din ls
```



Mulțimi

Accesare

- ▶ nu sunt indexate => NU `s[0]`, feliere etc
- ▶ parcurgere element cu element

```
for x in s:  
    print(x)
```

Muṭimi

Operatori

- ▶ `in, not in`

- ▶ `==, !=`

`{1,2,3} == {3,2,1}`

Mulțimi

Operatori

▶ $<$, \leq , $>$, \geq testează **incluziunea**

$\{1, 2\} < \{1, 2, 4\}$

$\{2, 4\} > \{2, 3\}$

Mulțimi

Operatori

► | reuniune

```
s1 = {5,7,10}; s2 = {5,10,15,20}; s3 = {8}
```

```
s = s1 | s2 | s3
```

Mulțimi

Operatori

- ▶ | reuniune

$s1 = \{5, 7, 10\}; s2 = \{5, 10, 15, 20\}; s3 = \{8\}$

$s = s1 \mid s2 \mid s3$

- ▶ & intersecție
- ▶ – diferența
- ▶ \wedge diferența simetrică

Mulțimi

Pentru incluziune și operații cu mulțimi – există și metode

- diferența: metodele pot primi un iterabil, operatorii doar set**

Mulțimi

Pentru incluziune și operații cu mulțimi – există și metode

- diferența: metodele pot primi un iterabil, operatorii doar set
- `issubset`, `issuperset`
- `union`, `difference`, `intersection`, `symmetric_difference` ...

Multipli

```
s1 = {5,7,10}
```

```
s2 = {5,10,15,20}
```

```
s3 = {8}
```

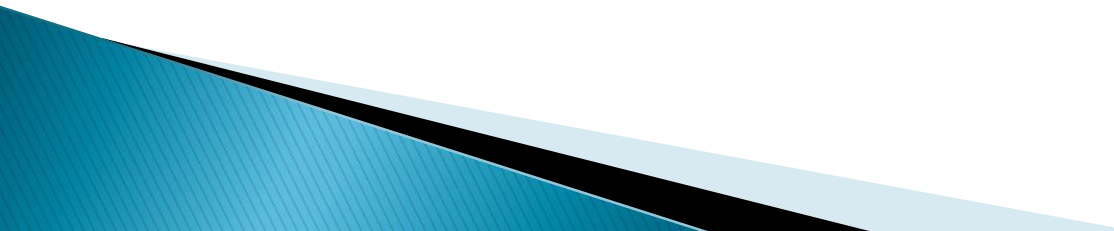
```
s4 = [5,6] #s4="ab"
```

```
s = s1.union(s2,s3,s4)
```

```
print(s1)
```

```
print(s)
```

```
#s=s1|s2|s3|s4
```



Mulțimi

Metode – modificare

- `add(element)` – adăugare element
- `update(iterabil_1[, ..., iterabil_n])`
–adauga elementele iterabililor primiți ca parametru
- `remove(element)` – eliminare element,
eroare dacă elementul nu este în mulțime
- `discard (element)` – eliminare element

Multi

```
s = {5,7,10}
```

```
s.add(11)
```

```
s.update({1,2})
```

```
s.update([3,1,10], "ab")
```

```
print(s)
```

```
s.remove(7)
```

```
s.discard(15) #s.remove(15)
```

```
print(s)
```



Mulțimi imutabile

Mulțimi imutabile

Clasa frozenset

- ▶ Aceleași metode ca la set, mai puțin cele care modifică mulțimea
- ▶ Creare folosind constructorul:

```
s1 = frozenset(iterabil)
```

```
svid = frozenset()
```



Mulțime imutabilă

► Clasa frozenset

```
s1 = frozenset([3,5,4,5])
```

```
s2 = {4,6}
```

```
s = s1 | s2
```

```
print(s,type(s)) #frozenset
```

```
s = s1.union(s2)
```

```
print(s,type(s)) #frozenset
```

```
s = s2 | s1
```

```
print(s,type(s)) #set
```

Dicționare

<https://docs.python.org/3.9/library/stdtypes.html#dict>

Dicționare

- Un **dicționar** – colecție de **perechi cheie și valoare** (fiecărei chei îi este asociată o valoare)
- Cheia – este unică și **imutabilă** + hash-uibilă => toate componentele cheii trebuie să fie imutabile
- ▶ Dicționarele sunt **mutabile**

Dictionare

```
t = (1,2) #poate fi cheie in dictionar
```

```
t = (1,[2,3]) #nu poate fi cheie in dictionar
```

Dicționare

- Indexate după cheie, nu după index:

`d[cheie] = valoarea asociata`

- Implementare internă – căutare eficientă (medie $O(1)$) după cheie (după codul hash asociat cheii)
=> v. tabele de dispersie

Dicționare

- Valoarea asociată cheii – orice tip
- Cheile pot avea tipuri diferite

Dictionare

Creare

```
d = {} #vid
```

```
d = {"a":2, "b":3}
```

#valorile - pot avea tip diferit

```
d = {"a":2, "b":3, "-":"semn"}
```

Dicționare

Creare

```
#valorile pot fi mutabile
```

```
d = {1:[2,3], 2:[1], 3:[1]}
```

Dictionare

Creare

```
#cheile trebuie sa fie hash-uibile
```

```
d = { (1,2): "tuplu", 3: "numar",  
      frozenset({2,4}): "frozen set" }
```

```
# d = { (1,2): "tuplu", 3: "numar",  
      {2,4}: "set" } #NU
```

Dictionare

Creare

▶ `dict(secventa_de_perechi_chei_valoare)`

```
d = dict([("a",1) , ("b",2)])
```

```
print(d)
```

```
d = dict(("a",1) , ("b",2))
```

```
print(d)
```



Dictionare

Creare

▶ `dict.fromkeys(iterabil_chei[,valoare_default])`

=> Toate valorile asociate cheilor sunt None sau
valoare_default (dacă aceasta se specifică)

```
d = dict.fromkeys("aeiou",0)
```

```
print(d)
```



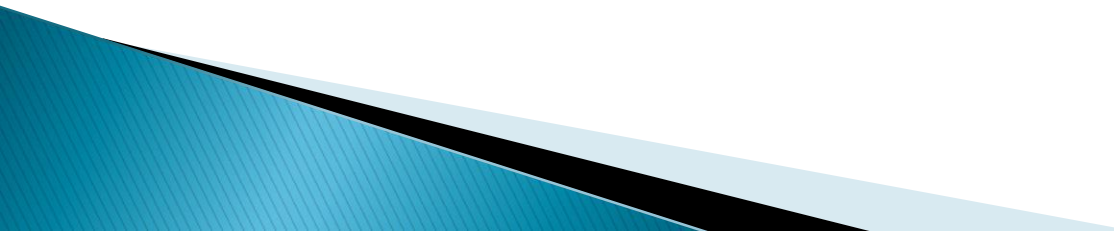
Dictionary

Creare

- Comprehensiune

```
d = {x:0 for x in "aeiou"} # dict.fromkeys  
print(d["e"])
```

```
d = {x:chr(ord(x)+1) for x in "aeiou"}  
print(d["e"])
```



Dicționare

Accesare elemente

- `d[cheie]` => **eroare** dacă nu există cheia
- `d.get(cheie[,valoare_default])`
=> returnează valoarea asociată cheii; dacă cheia nu există returnează **None** sau, dacă este specificată, `valoare_default` dată ca al doilea parametru
- `d.setdefault` – discutată la actualizare

Dictionare

```
d = {"a":2, "b":3}

print(d["a"])

# print(d["c"]) #KeyError: 'c'

print(d.get("c")) #None

print(d.get("c",0)) #0
```

Dicționare

Actualizare

- `d[cheie] = valoare` – inserează dacă nu există cheia
- `d.setdefault(cheie[, valoare]) =>` inserează în dicționar cheia dată dacă nu există, cu valoarea `None` sau cea specificată la al doilea parametru și **returnează valoarea cheii (existentă sau tocmai inserată)**

Dictionare

```
d = {"a":2, "b":3 }
```

```
x = d.setdefault("c",0)
```

```
print(x,d)
```



Dictionare

```
d = {"a":2, "b":3 }
```

```
x = d.setdefault("c",0)
```

```
print(x,d)
```

```
x=d.setdefault("c",10) #exista, nu se actualizeaza
```

```
print(x,d)
```



Dictionare

```
d = {"a":2, "b":3 }
```

```
x = d.setdefault("c",0)
```

```
print(x,d)
```

```
x=d.setdefault("c",10) #exista, nu se actualizeaza
```

```
print(x,d)
```

```
d["c"] = 7 #se actualizeaza
```

```
print(d)
```

```
d["e"] = 8 #se insereaza
```



Dicționare

Actualizare

- `d.update(dictionar)`
- `d.update(iterabil cheie-valoare)` =>
reuniune de dicționare, cu actualizarea cheilor care există deja

```
d = {"a":2, "b":3 }
```

```
d.update({"b":4, "altceva":0})
```

```
d.update([("f",5), ("g",4)])
```

```
print(d)
```

Dicționare

Actualizare

- `d.pop(cheie[,valoare]) =>` elimină cheia (cu valoarea asociată) și returnează valoarea asociată; dacă cheia nu există dă **eroare** sau returnează **valoarea furnizată la al doilea parametru**
- `del d[cheie]`
- `d.clear()`

Dictionare

```
d = {"a":2, "b":3 , "c":4}
```

```
#d.pop("A") #eroare KeyError: 'A'
```

```
x=d.pop("a")
```

```
print(x)
```

```
print(d)
```

```
x=d.pop("A", 0)
```

```
print(x)
```

```
print(d)
```

```
del d["b"] #ca si pop
```



Dicționare

Accesare + Parcurgere

- `d.keys()` => chei (un view care se updateaza automat odata cu schimbarea dictionarului *d*)
- `d.values()` => valori
- `d.items()` => tupluri (cheie, valoare)

Dictionary

```
d = {"a":2, "b":3, "-":7}

print(d.keys(), type(d.keys())) #tip dict_keys
print(d.values(), type(d.values()))
print(d.items(), type(d.items()))

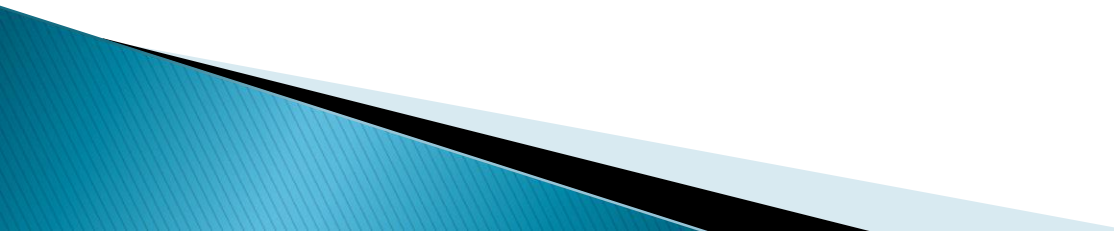
ls_key=d.keys()

print(ls_key)

d.update({"c":5, "e":2, "a":4})

print(d)

print(ls_key)
```



Dictionare

```
d = {"a":2, "b":3, "-":7}
```

```
for x in d: #dupa cheie
```

```
    print(x, d[x])
```

```
for p in d.items():
```

```
    print(p, type(p))
```

```
for p in d.values():
```

```
    print(p, type(p))
```



Dictionare

```
#ce face secventa de cod?
```

```
d = {"a":2, "b":3, "c":7}
```

```
d1 = {x:d[x] for x in d.keys()-{"b"}}
```

```
print(d1)
```



Dictionare

Operatori

- `in`, `not in` (după cheie)
- `==` , `!=`

Metode comune

- `max`, `min` – pentru cheie
- `len`

Dictionare

```
d1 = {"a":2, "b":3 , "c":4}
```

```
d2 = {"b":3 , "a":2, "c":4}
```

```
print(d1 == d2)
```

```
print(len(d1))
```

```
print(max(d1))
```



Dicționare

Exemplul 1 – Frecvența caracterelor dintr-un text dat pe o linie



Dicționare

Exemplul 2 – se dă o listă de n puncte în plan prin coordonate și etichetă. Dacă un punct apare de mai multe ori în listă se păstrează ultima etichetă asociată lui. Se citește un nou punct (x,y) . Să se afișeze eticheta acestuia, dacă a fost dată.

5

1 2 punctul 1

1 3 punctul 2

2 5 punctul 3

1 2 punctul 1 nou

4 1 punctul 4

3 2

Fişiere

Fișiere text

- ▶ Fișier text=secvență de caractere organizată pe linii și stocată pe un suport de memorie extern

Fișiere

► Deschiderea unui fișier text

```
f = open("cale_fisier")
```

```
f = open("cale_fisier", "mod_de_deschidere")
```

unde `mod_de_deschidere` poate fi de exemplu:

- `"r"` - `read` = pentru citire (eroare dacă nu există)
- `"w"` - `write` = pentru scriere (suprascrie dacă există deja)
- `"a"` - `append` = pentru scriere la sfârșit (dacă nu există se creează)

Fișiere

- ▶ Închiderea unui fișier text

```
f.close()
```

Fișiere

▶ Deschiderea unui fișier text

```
with open("cale fisier") as f:
```

– cu with nu mai trebuie închis fișierul cu f.close()

Fișiere

► Citirea din fișier text

- `s = f.read()` – tot fișierul într-un șir de caractere
- `s = f.readline()` – linia curentă într-un șir de caractere (cu `\n` la sfârșit)
- `l = f.readlines()` – toate liniile într-o listă de șiruri de caractere (incluzând sfârșiturile de linie)

Fişiere

```
f = open('date.in')  
linie = f.readline()  
print(repr(linie))  
#tot = f.read()  
#print(tot)  
vector_linii=f.readlines()  
print(vector_linii)  
f.close()
```

Fişiere

```
"""date.in
```

```
3 4 10 80
```

```
5 18
```

```
"""
```

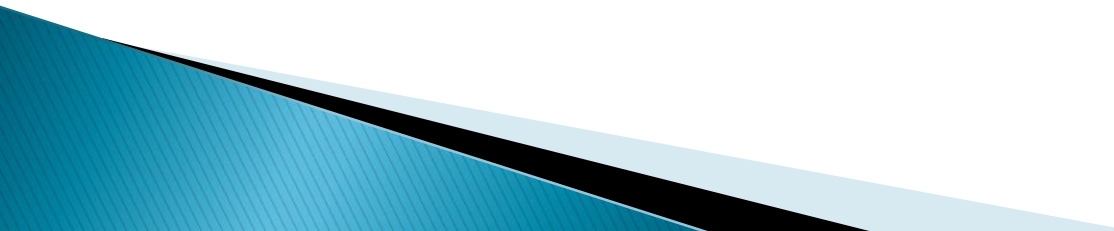
```
with open('date.in', 'r') as f:
```

```
    linie = f.readline()
```

```
    while linie != '':
```

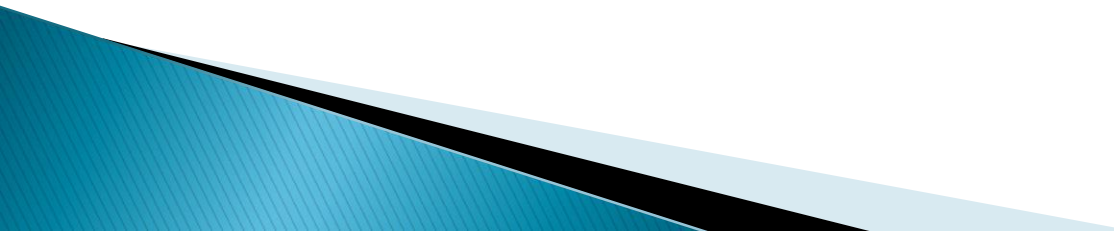
```
        print(linie, end='') #linie continue \n
```

```
        linie = f.readline()
```



Fișiere

```
"""date.in  
3 4 10 80  
5 18  
"""  
  
#citire linie cu linie  
with open('date.in', 'r') as f:  
    linie = f.readline()  
    while linie != '':  
        print(linie, end='') #linie continue \n  
        linie = f.readline()
```



Fișiere

```
"""date.in
```

```
3 4 10 80
```

```
5 18
```

```
"""
```

```
#citire linie cu linie
```

```
with open('date.in', 'r') as f:
```

```
    for linie in f:
```

```
        print(linie, end='') #linie continue \n
```

Fişiere

```
"""date.in
```

```
3 4 10 80
```

```
5 18
```

```
"""
```

```
f = open("date.in")
```

```
ls1 = f.readline().split() #include si \n
```

```
print(l1)
```

```
ls2 = f.readline().split()
```

```
print(l2)
```

```
f.close()
```



Fișiere

- ▶ Scrierea în fișier text

```
f.write(sir_de_caractere)
```

```
f.writelines(colectie_de_siruri) - nu  
adaugă automat sfârșitul de linie
```

```
with open("date.out", "w") as g:
```

```
g.writelines(["linia 1\n", "linia 2", "3"])
```



Fişiere

► Copiere

```
with open("date.in", 'r') as f,  
open("date.out", 'w') as g:  
    linii = f.read()  
    g.write(linii)
```


Fişiere

► Copiere

```
with open("date.in", 'r') as f,  
open("date.out", 'w') as g:  
    linii = f.readlines()  
    g.writelines(linii)
```

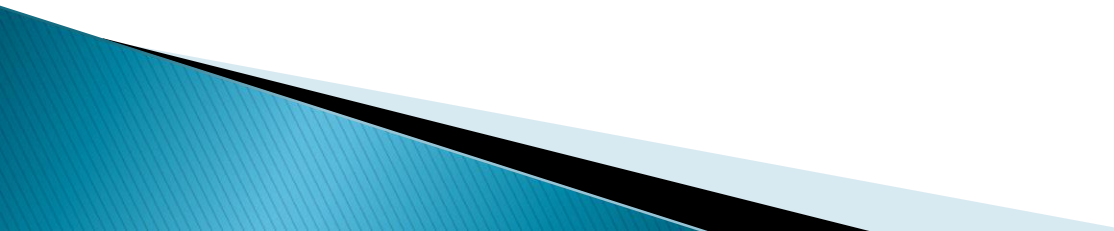
Fișiere

- ▶ Fișierul standard de intrare – citire șir cu pe mai multe linii

```
import sys
```

```
strofa = sys.stdin.read()
```

```
print(strofa)
```



Fișiere

- ▶ Fisierul standard de intare – citire linii

```
linie = input()
```

```
while len(linie)>0:
```

```
    print(linie)
```

```
    linie = input()
```

Fișiere

- ▶ Fisierul standard de intare – citire linii

```
for linie in sys.stdin:
```

```
    #print(linie,end=" ")
```

```
    sys.stdout.write(linie)
```



Fișiere

► Exercițiu (similar seminar/laborator)

Se citește din fișierul "date.in" un șir de numere naturale separate prin spațiu. Să se afișeze în fișierul "date.out" media aritmetică a numerelor din șir. Mai exact, rezultatul va fi un șir de forma:

"(nr1 + ...nr_k)/n=media", unde nr1, ..., nrk sunt numerele care apar în șir și media este afișată cu 2 zecimale

Complexitate operații secvențe



Complexitate operații

- ▶ **list - intern – vector**
- ▶ **set - tabel dispersie**
- ▶ **dict - tabel dispersie**

<https://wiki.python.org/moin/TimeComplexity>

Complexitate operații

► list

Operation	Average Case	<u>Amortized Worst Case</u>
Copy	$O(n)$	$O(n)$
Append	$O(1)$	$O(1)$
Pop last	$O(1)$	$O(1)$
Pop(i)	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(k+n)$
Extend	$O(k)$	$O(k)$
<u>Sort</u>	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
x in s	$O(n)$	
min(s), max(s)	$O(n)$	
Get Length	$O(1)$	$O(1)$

Complexitate operații

► dict

Operation	Average Case	Amortized Worst Case
k in d	$O(1)$	$O(n)$
Copy	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(n)$
Set Item	$O(1)$	$O(n)$
Delete Item	$O(1)$	$O(n)$
Iteration	$O(n)$	$O(n)$

Complexitate operații

► set

Operation	Average case	Worst Case
$x \text{ in } s$	$O(1)$	$O(n)$
Union $s t$	<u>$O(\text{len}(s) + \text{len}(t))$</u>	
Intersection $s \& t$	$O(\min(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$
Difference $s - t$	$O(\text{len}(s))$	
Symmetric Difference	$O(\text{len}(s))$	$O(\text{len}(s) * \text{len}(t))$

Structuri de date – implementări Python



Stivă

- ▶ list

Coadă

- ▶ list – lent pop NU
- ▶ collections.deque
- ▶ queue.Queue – sincronizata, mai lenta

Coadă de prioritati

- ▶ Operația fundamentală– extragere minim/maxim
- ▶ `heapq`: heap (binar)
- ▶ `queue.PriorityQueue`: – **sincronizata, mai lenta**

