Alte tipuri de colecții - mulțimi, dicționare

ALTE TIPURI DE COLECȚII

Pe lângă secvențe, care sunt colecții indexate, există și alte tipuri de colecții pe care le vom prezenta în continuare: mulțimi, dicționare. Pentru a înțelege modul în care acestea funcționează sau sunt memorate, vom prezenta întâi conceptele de funcție hash (de dispersie) și tabel de dispersie.

Hashcode - Funcții de dispersie. Tabele de dispersie

Unele obiecte au asociate un cod numeric numit **cod (valoare) de dispersie** sau **cod hash**, a cărui utilitate o vom prezenta în continuare.

Un obiect se numește *hash-uibil* dacă are un cod numeric **hash asociat**, returnat de metoda **`__hash__()`** (= **cod de dispersie**, **valoare hash), care este** 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__()`**)

Exemplu

```
t = (1,2)
print(hash(t))  #print(t.__hash__())
t = (1,[2,3])
print(hash(t)) # print(t.__hash__())  #TypeError: unhashable type: 'list'
s = "testare"
print(s, s.__hash__())
s = s[:len(s)-1]
print(s, s.__hash__())
s = s + "e"
print(s, s.__hash__())
```

Astfel, unui obiect de dimensiune arbitrară i se asociază un număr întreg (de dimensiune fixă), de aceea pot apărea coliziuni (obiecte diferite cu același număr asociat).

Codul hash asociat unui obiect poate fi folosit pentru structuri de dare indexate după chei. Spre exemplu, dacă vrem să sortăm o listă de numere naturale mai mici decât 100, putem folosi un vector de frecvență **`v`** în care **`v[i]`** = numărul de apariții ale lui i în listă. Dacă însă lista este de numere mari, sau de șiruri de caractere, nu putem folosi elementele din listă ca index pentru vectorul **`v`**; dacă însă fiecare element din listă are asociat un cod numeric, putem folosi acel cod ca index în structuri de date indexate după chei care generalizează ideea de vector de frecvențe, spre exemplu **table de dispersie**. Aceasta este una dintre utilitățile codului hash asociat unui obiect.

Tabele de dispersie

Un tabel de dispersie este structură de date pentru căutare eficientă după chei hash-uibile. Fiecărei poziții `c` din tabel îi corespunde un bucket (o listă) cu obiectele având indexul asociat codului hash (modulo numărul de bucketuri) egal cu c; pot fi mai multe obiecte cu același index.

Funcția hash trebuie aleasă astfel încât să se **minimizeze** numărul coliziunilor (chei diferite care produc aceleași hash-uri). Coliziunile apar în mod inerent, deoarece lungimea maximă a hash-ului este fixă, iar obiectele de stocare pot avea lungimi și conținut arbitrare. În cazul apariției unei coliziuni, valorile se stochează pe aceeași poziție - în același **bucket**. În acest caz, căutarea se va reduce la compararea valorilor efective în cadrul bucket-ului.

Căutarea unui obiect în tabel se face, în mare, astfel:

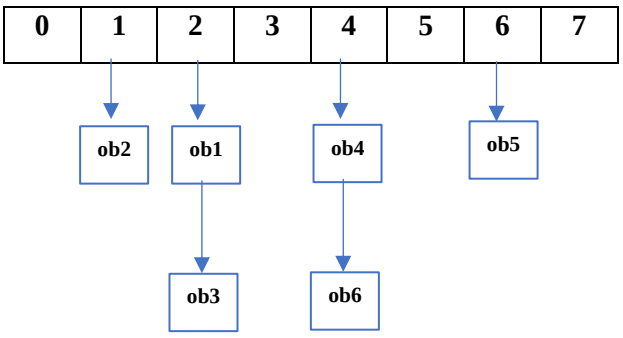
- se determină c = indexul asociat codului hash al obiectului, se accesează bucketul c
- se caută în acest bucket obiectul (folosind pentru testare metoda `__eq__()`).

Astfel, căutarea unei chei se face **în medie** timpul O(1) (dar defavorabil O(n), din cauza coliziunilor posibile; de aceea este important ca funcția de dispersie să furnizeze valori cât mai uniform distribuite, pentru ca dimensiunea unui bucket să fie cât mai mică)

Exemplu.

Să presupunem că avem la dispoziție 3 biți (deci 2^3=8 bucketuri) pentru memorarea codului hash și avem la dispoziție 6 obiecte, cu codurile hash asociate (modulo numărul de bucketuri) date în tabelul de mai jos. Tabelul de dispersie poate fi vizualizat ca în desenul alăturat

Obiect	Hash code (modulo numărul de bucketuri)
`ob1 = "a"`	2
`ob2 = "bc"`	1
`ob3 = "ad"`	2
`ob4 = "m"`	4
`ob5 = "casa"`	6
`ob6 = "aab"`	4



Pentru a căuta cheia `"ad"` în tabelul de dispersie, se calculează codul hash al ei (modulo numărul de bucketuri), în acest caz `2`, se accesează bucketul `2` și se caută valoarea în bucket.

O cheie poate avea asociată o valoare, ca în cazul dicționarelor (spre exemplu cheia poate fi un cuvânt iar valoarea asociată să fie frecvența cu care apare într-un fișier).

MULȚIMI

Despre mulțimi - clasa set. Crearea unei mulțimi

O mulțime este o colecție de elemente distincte (în care valorile nu se repetă).
O mulțime **nu este indexată** (elementele nu au indici, ca în cazul șirurilor de caractere sau al listelor).

Elementele unei mulțimi nu se **păstrează neapărat în ordinea în care au fost inserate.**

O mulțime poate conține elemente de tipuri diferite, dar trebuie să fie "***imutabile*", de fapt *hash-uibile*** (o mulțime fiind memorată intern cu structuri de tipul tabelelor de dispersie).

Creare

Elementele unei mulțimi se dau cu virgulă și sunt cuprinse între acolade. Mulțimea vidă însă nu se poate crea printr-o pereche de acolade (astfel se creează un dicționar vid, nu o mulțime), ci cu ajutorul funcției (constructorului) **`set()`**.

Exemplu.

```
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 , deoarece [1,2] este de tip list,
#nu este hash-uibil (o listă este mutabilă)
#TypeError: unhashable type: 'list'
```

Pentru a crea o mulțime se poate folosi și constructorul **`set([iterabil])`**. Apelul constructorului fără parametri **`set()`** va returna mulțimea vidă.

Folosind funcția **`set()`** pentru secvențe obținem mulțimea elementelor distincte din secvență (elementele distincte dintr-o listă, caracterele distincte dintr-un șir de caractere)

Exemplu.

```
ls = [2, 3, 1, 3, 2, 6]
s = set(ls) #elementele distincte
cuv = "aceeasi"
s = set(cuv)
```

Exercițiu – Ce afișează programul următor?

```
s = set() #multimea vida
s1 = set("multime")
s2 = set(["multime"])
s3 = set(("multime"))
print(s,s1,s2,s3)
```

Comprehensiune

Mulțimile se pot crea și folosind comprehensiune (secvențe de inițializare).

Exemple:

1. Mulțimea cifrelor:

```
s = {x for x in range(10)}
```

2. Elementele distincte pozitive dintr-o listă ls:

```
ls = [2, 3, 1, 3, -2, 6, 2]
s = {x for x in ls if x>0}
```

Accesarea elementelor. Parcurgere

După cum am amintit deja, mulțimile sunt colecții neindexate, deci elementele nu se pot accesa prin indici sau prin feliere, ca în cazul listelor, tuplurilor și șirurilor. O mulțime se poate însă parcurge cu **for**, ca orice obiect iterabil:

```
s = set("alfabetar")
for x in s:
    print(x)
```

Ordinea în care sunt afișate elementele nu coincide neapărat cu cea în care au fost adăugate elementele.

Operatori și funcții uzuale

Pentru mulțimi se pot folosi următorii operatori, deja cunoscuți:

- **len**, **min**, **max** (!!nu și index sau count)
- **in, not in**
- **==, !=**

  Exemplu: Expresia `{1,2,3} == {3,2,1}` este **True**

- Operatorii relaționali **<**, **<=**, **>**, **>=** testează incluziunea

  Exemplu: Expresia `{4,2} < {1,2,4}` este adevărată, prima mulțime fiind inclusă în a doua, iar expresia `{2,4} > {2,3}` este **False**

- Operatori pentru operații cu mulțimi: `|` pentru reuniune, `&` pentru intersecție, `-` pentru diferență, `^` pentru diferența simetrică

  Operatorii pentru operații se pot înlănțui.

  Exemplu:

```
s1 = {5,7,10}; s2 = {5,10,15,20}; s3 = {8}
s = s1 | s2 | s3  #reuniunea celor 3 multimi
s = s1 ^ s2  #diferenta simetrica s = (s1 | s2)  - (s1 & s2)
```

- Operatori de atribuire corespunzători operatorilor pentru operații cu mulțimi:
  `|=, &= , ^=, -=`

  Exemplu:

```
s1 = {5,7,10}; s2 = {5,10,15,20}
s1 |= s2
print(s1)
```

Operații cu mulțimi

Pentru a testa incluziunea dintre două mulțimi și a efectua operații cu mulțimi există, pe lângă operatorii amintiți în secțiunea precedentă, și metode. Diferența între operatori și metodele corespunzătoare este că metodele pot primi ca parametru orice iterabil, operatorii pot fi folosiți doar pentru operanzi de tip mulțime.

- **`issubset`**, **`issuperset`** – testează dacă mulțimea curentă este submulțime pentru colecția primită ca parametru, respectiv dacă o include

  Exemplu

```
s1 = {3, 8}
s2 = [3, 6, 8, 12]
print(s1.issubset(s2))
#print(s1<=s2) #TypeError: '<=' not supported between instances
of 'set' and 'list'
```

- **`union`**, **`difference`**, **`intersection`**, **`symmetric_difference`** - metode pentru operații cu mulțimi, **returnează mulțimea obținută** (nu se modifică obiectul care apelează metoda, ci se returnează un obiect nou; metodele metode pentru operații cu mulțimi prin care se poate modifica mulțimea sunt prezentate în secțiunea următoare)

  Exemplu

```
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 - eroare daca s4="ab"
```

Metode pentru modificarea unei mulțimi

- **`s.add(elem)`** – adaugă elementul **`elem`** la mulțimea
- **`update(iterabil_1[,...,iterabil_n])`** – adaugă la mulțime elementele iterabililor primiți ca parametru (reuniune cu modificarea mulțimii, ca și **`|=`**)

- **`intersection_update, difference_update, symmetric_difference_update`** – ca și **`update`**, sunt **metode de actualizare a mulțimii corespunzătoare celorlalte operații cu mulțimi**
- **`remove(elem)`** – elimină din mulțime elementul primit ca parametru, cu **eroare** dacă elementul nu este în mulțime
- **`discard(elem)`** – ca și metoda **`remove`** elimină din mulțime elementul primit ca parametru, dacă este în mulțime; spre deosebire de **`remove`** nu aruncă eroare dacă elementul nu este în mulțime.

Exemplu

```
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)
```

Implementare internă. Complexitatea operațiilor

Un obiect de tip set este memorat intern cu un mecanism similar tabelelor de dispersie, de aceea complexitatea operațiilor principale este cea dată de următorul tabel (sursa: https://wiki.python.org/moin/TimeComplexity):

Operation	Average case	Worst Case
`x in s`	`O(1)`	`O(n)`
`Union s|t`	`O(len(s)+len(t))`	
`Intersection s&t`	`O(min(len(s), len(t))`	`O(len(s) * len(t))`
`Difference s-t`	`O(len(s))`	
`Symmetric Difference`	`O(len(s))`	`O(len(s) * len(t))`

MULȚIMI IMUTABILE

Pentru a crea mulțimi imutabile (în care nu se mai pot adăuga sau elimina elemente) în Python există clasa **`frozenset`**. O astfel de mulțime este necesară, spre exemplu, când este nevoie să indexăm un dicționar după chei care sunt mulțimi, deoarece cheile pentru un dicționar trebuie să fie hash-uibile deci, în general, imutabile (deci nu putem folosi obiecte de tip **`set`** sau **`list`** ca și chei; vom discuta în detaliu despre dicționare în secțiunea următoare).

O mulțime imutabilă se poate crea folosind constructorul **`frozenset([iterabil])`**

Exemplu:

```
s = frozenset([3,5,4,5])
s_vid = frozenset() #multimea vida ca frozenset
print(s,s_vid)
```

Pentru mulțimi imutabile se pot folosi aceeași operatori și aceleași metode ca pentru mulțimi de tip set, mai puțin cele care modifică mulțimea.

```
s1 = frozenset("programarea")
s2 = frozenset("algoritmilor")
print("litere comune", s1&s2)
```

Operațiile pe mulțimi se pot aplica și între un obiect de tip **`set`** și unul de tip **`frozenset`**, tipul rezultatului fiind dat de tipul variabilei cu care se apelează metoda sau, în cazul operatorilor, de tipul primului operand.

Exemplu:

```
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

Despre dicționare – clasa dict. Creare

Un **dicţionar** este o colecție de **perechi cheie şi valoare** (fiecărei chei îi este asociată o valoare), spre exemplu perechi de forma (șir de caractere, frecvență în fișier).
Pentru a memora un dicționar în Python există clasa **`dict`**:

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

Dicționarele sunt **mutabile, se pot actualiza**

Cheile dintr-un dicționar pot avea tipuri diferite, la fel și valorile. O cheie dintr-un dicționar este unică și **imutabilă** (+ hash-uibilă); toate componentele cheii trebuie să fie imutabile. Spre exemplu:

- **`t = (1,2)`** poate fi cheie în dicționar
- **`s ="un sir"`** poate fi cheie în dicționar
- **`ls = [2, 3]`** nu poate fi cheie în dicționar
- **`t = (1,[2,3])`** nu poate fi cheie în dicționar

Valorile pot avea orice tip.

Dicționarele sunt indexate după cheie, nu după index; putem accesa valoarea asociată unei chei cu apelul **`d[cheie]`**

Dicționarele sunt implementate intern astfel încât să permită o căutare eficientă după cheie (folosind codul hash asociat cheii, timpul mediu de căutare fiind medie O(1) - v. Tabele de dispersie)

Creare

Perechile (cheie, valoare) dintr-un dicționar se pot specifica între acolade, separate prin virgulă, sub forma **`cheie:valoare`**, spre exemplu:

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

#valorile - pot avea tip diferit, la fel si cheile:
d = {0:0, "a":2, "b":3,  "-":"semn"}

#valorile pot fi mutabile:
d = {1:[2,3], 2:[1], 3:[1]}

#cheile trebuie sa fie hash-uibile
d = {(1,2):"tuplu", 3:"numar", frozenset({2,4}): "multime imutabila"}
# d = {(1,2):"tuplu", 3:"numar",  {2,4}: "multime"} #NU, deoarece
# o mulțime de tip set este mutabilă
```

Un dicționar se poate crea și folosind constructorul **dict():**

```
# dict de secventa de perechi
d = dict([("unu", 1), ("doi", 2), ("trei", 3)])
d = dict((("unu", 1), ("doi", 2), ("trei", 3)))

# argumente cheie=valoare
d = dict(unu = 1, doi = 2, trei = 3)

# mixt
d = dict([("unu", 1), ("doi", 2)], trei=3)
```

Putem crea un dicționar cu o listă de chei dată și o valoare default pentru toate cheile folosind metoda **fromkey**:

```
dict.fromkey(iterabil_chei[,valoare_default])
```

Toate valorile asociate cheilor vor fi **None** sau, dacă este specificată, **valoare_default** dată ca al doilea parametru:

```
d = dict.fromkeys("aeiou") #valori asociate None
d = dict.fromkeys("aeiou",0) #valorile 0
```

Comprehenisune

Dicționarele se pot crea și prin comprehensiune (secvențe de inițializare), spre exemplu:

1. un dicționar de tipul celui creat cu metoda **fromkeys** se poate crea astfel:

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

2. un dicționar având ca și chei vocalele și ca valoare litera următoare din alfabet:

```
# perchi de forma (vocala, litera urmatoare din alfabet
d = {x:chr(ord(x)+1) for x in "aeiou"}
print(d["e"])
```

3. un dicționar cu frecvența cuvintelor într-o listă (cheia este cuvântul și valoarea este frecvența):

```
ls = ["o", "lista", "o", "secventa", "o", "lista", "de", "cuvinte"]
#set(ls) = multimea cuvintelor distincte din ls
d = {x:ls.count(x) for x in set(ls)}
print(d)
```

Accesarea valorii asociate unei chei

Valoarea asociată unei chei de poate accesa cu apelul:

```
d[cheie]
```

Dacă cheia nu există în dicționar, se va arunca eroarea `KeyError`

O altă modalitate de a accesa valoarea asociată unei chei, care însă returnează o valoare default dacă cheia nu există, este dată de metoda get:

```
d.get(cheie[,valoare_default])
```

Metoda returnează valoarea asociată cheii; dacă cheia nu există returnează **None** sau, dacă este specificată, **valoare_default** dată ca al doilea parametru.

Exemplu.

```
d = {"a":2, "b":3}
print(d["a"])
# print(d["c"]) #KeyError: 'c'
print(d.get("c")) #None
print(d.get("c",0)) #0
```

Actualizare

Pentru a modifica valoarea asociată unei chei sau a insera o pereche (cheie, valoare) se pot folosi metode similare celor de la accesarea unui element.
Astfel, cu ajutorul unei atribuiri de forma:

```
d[cheie] = valoare
```

se actualizează valoarea asociată cheii sau se inserează perechea (cheie, valoare), dacă nu există cheie în dicționar (!nu dă eroare, ca la interogare, dacă nu există cheia, ci o adaugă):

```
d = {"a":2, "b":3 }
d["c"] = 7 #se actualizeaza
print(d)
d["e"] = 8  #se insereaza
```

În clasa **dict** există și o serie de metode care permit actualizarea dicționarului:

- **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)**

  Exemplu

  ```
  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.update(dictionar), d.update(iterabil cheie-valoare)** – se reunesc dicționarele d și cel primit ca parametru, **cu actualizarea cheilor comune**, care există deja în **d**

  Exemplu

  ```
  d = {"a":2, "b":3 }
  d.update({"b":4,"altceva":0})
  d.update([("f",5),("g",4)])
  print(d)
  ```

- **`d.pop(cheie[,valoare])`** – se elimină cheia cheie (cu valoarea asociată) și returnează valoarea asociată cheii eliminate; dacă cheia nu există, metoda returnează valoarea furnizată la al doilea parametru sau dă eroarea **`KeyError`** dacă această valoare nu este furnizată:

  Exemplu

  ```
  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[cheie]`** – similar cu **`pop`**, dar dacă cheia nu există se aruncă eroarea **`KeyError`**

  Exemplu

  ```
  d = {"doi":2, "trei":3 , "patru":4}
  del d["doi"] #ca si pop
  print(d)
  #del d["DOI"] #eroare KeyError: 'DOI'
  ```

- **`d.clear()`** – șterge toate elementele dicționarului

Accesare chei si valori. Parcurgere

În clasa **`dict`** există metode care returnează atât o secvență cu toate cheile din dicționar, dar și cu toate valorile, sau o secvență de perechi (cheie, valoare) cu toate elementele din dicționar:

- **`d.keys()`** – returnează mulțimea de chei (de fapt un view care se actualizează automat odată cu schimbarea dicționarului d)
- **`d.values()`** - returnează „lista” de valori (view)
- **`d.items()`** - returnează „lista” de tupluri (cheie,valoare) (view)

Exemplu

```
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)
```

Astfel, putem itera cu for cheile dintr-un dicționar, sau valorile, sau perechi (cheie,valoare). Iteratorul **implicit** iterează secvența cheilor:

```
d = {"a":2, "b":3,"-":7}
for x in d: #implicit dupa cheie
    print(x, d[x])
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))
```

Pentru mulțimile (view) returnate de aceste metode se pot folosi operatori de mulțimi și metodele comune (**len**, **min**, **max**, **sorted**).

Exemple

1. Afișarea cheilor comune pentru două dicționare:

```
d = {"doi":2, "trei":3 , "patru":4}
d2 = {"doi":2, "cinci":5 , "patru":4}
print(d.keys() & d2.keys())
```

2. Crearea unui dicționar nou cu perechile (cheie, valoare) din d cu excepția cheii b

```
d = {"a":2, "b":3,"c":7}
d1 = {x:d[x] for x in d.keys()-{"b"}}
print(d1)
```

Operatori

- **in, not in** - testul de apartenență se face pentru chei:

  Exemplu

  ```
  d = {"doi":2, "trei":3 , "patru":4}
  print("doi" in d) #True
  print(2 in d) #False
  ```

- **== , !=** testează dacă două dicționare au aceeași mulțime de perechi (cheie, valoare)

  Exemplu

  ```
  {"doi":2, "trei":3, "patru":4} == {"patru":4, "doi":2, "trei":3}
  ```

  este **True**

Metode comune

- **max, min** – pentru mulțimea cheilor (!dacă sunt comparabile)
- **len**

Exemplu

```
d1 = {"a":2, "b":3 , "c":4}
print(len(d1))
print(max(d1))
print(max(d1.values()))
```

Exercițiu – Frecvența caracterelor dintr-un text dat pe o linie

```
s = input()
d1 = {}
for x in s:
    if x in d1:
        d1[x] += 1
    else:
        d1[x] = 1
print(d1)
```

```
#varianta 2
d3 = {}
for y in s:
    d3[y]=d3.get(y, 0) + 1
print(d3)
```

```
#varianta 3
#d2 = {x:0 for x in set(s)}
d2=dict.fromkeys(set(s),0)
for y in s:
    d2[y] = d2[y]+ 1
print(d2)
```

Exercițiul – Temă 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.
Exemplu de date de intrare:
5
1 2 punctul 1
1 3 punctul 2
2 5 punctul 3
1 2 punctul 1 nou
4 1 punctul 4
1 2
(se va afișa **punctul 1 nou)**

Implementare internă. Complexitatea operațiilor

Un obiect de tip dict este memorat intern cu un mecanism similar tabelelor de dispersie, de aceea complexitatea operațiilor principale este cea dată de următorul tabel (sursa: https://wiki.python.org/moin/TimeComplexity):

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)**

Un articol despre memorarea dicționarelor: https://medium.com/codex/internal-implementation-of-dictionary-in-python-5b739d5535a4