

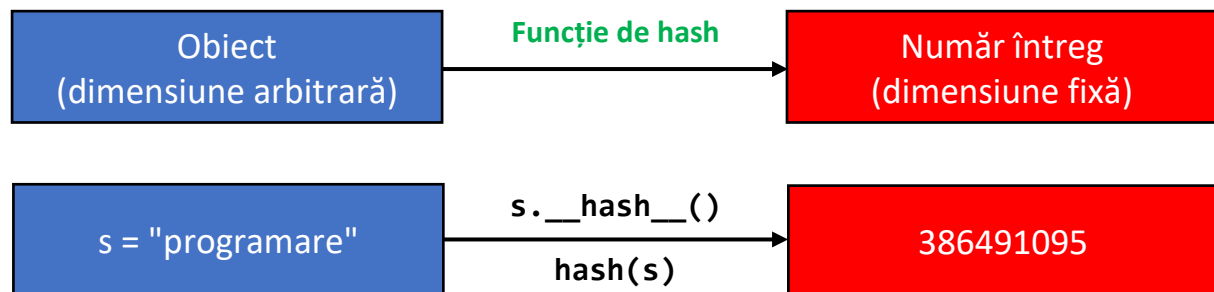
CURS 5

Colecții de date

Tabele de dispersie (hash table)

În general, o *funcție de dispersie* (*hash function*) este un algoritm care asociază unui șir binar de lungime arbitrară un șir binar de lungime fixă numit *valoare de dispersie* sau *cod de dispersie* (*hash value*, *hash code* sau *digest*).

În limbajul Python, clasele corespunzătoare tipurilor de date imutabile (i.e., clasele `int`, `float`, `complex`, `bool`, `str`, `tuple` și `frozenset`) conțin metoda `__hash__()` care furnizează valoarea de dispersie asociată unui anumit obiect sub forma unui număr întreg pe 32 sau 64 de biți.



Alternativ, valoarea de dispersie a unui obiect poate fi aflată utilizând funcția predefinită `hash(obiect)`.

```

s = "Ana are mere!"
print(f"hash('{s}') = {s.__hash__()}")
print(f"hash('{s}') = {hash(s)}\n")

n = 12345
print(f"hash({n}) = {hash(n)}\n")

n = 2*100
print(f"hash({n}) = {hash(n)}\n")

n = 3.14
print(f"hash({n}) = {hash(n)}\n")

t = (1, 2, 3, 4, 5)
print(f"hash({t}) = {hash(t)}\n")
  
```

```

C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python
hash('Ana are mere!') = -2312829570296290796
hash('Ana are mere!') = -2312829570296290796

hash(12345) = 12345

hash(1267650600228229401496703205376) = 549755813888

hash(3.14) = 322818021289917443

hash((1, 2, 3, 4, 5)) = -5659871693760987716
  
```

În limbajul Python, orice funcție de dispersie trebuie să satisfacă următoarele două condiții:

1. două obiecte care sunt egale din punct de vedere al conținutului trebuie să fie egale și din punct de vedere al valorilor de dispersie (i.e., dacă `obiect_1 == obiect_2` atunci obligatoriu și `hash(obiect_1) == hash(obiect_2)`);

2. valoarea de dispersie a unui obiect trebuie să rămână constantă pe parcursul executării programului în care este utilizat obiectul respectiv, dar nu trebuie să rămână constantă în cazul unor rulări diferite ale programului.

Putem observa faptul că ambele condiții de mai sus sunt respectate de funcția predefinită `hash`, rulând de mai multe ori următoarea secvență de instrucțiuni:

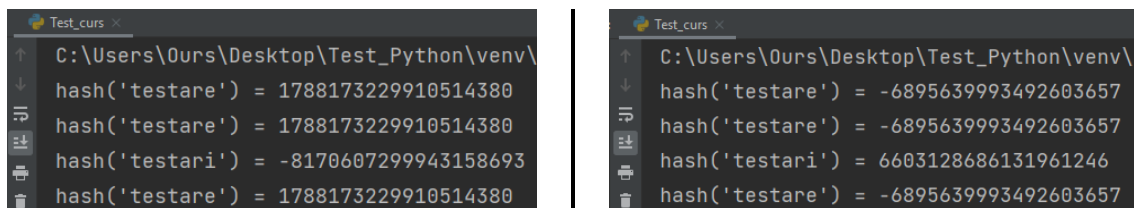
```
s = "testare"
print(f"hash('{s}') = {hash(s)}")

t = "test"
t += "are"
print(f"hash('{t}') = {hash(t)}")

s = s[:len(s)-1] + "i"           #s = "testari"
print(f"hash('{s}') = {hash(s)}")

s = s[:len(s)-1] + "e"           #s = "testare"
print(f"hash('{s}') = {hash(s)}")
```

Astfel, vom observa faptul că la fiecare rulare a secvenței primele două valori afișate și ultima sunt întotdeauna egale, fără a rămâne constante în cazul mai multor rulări:



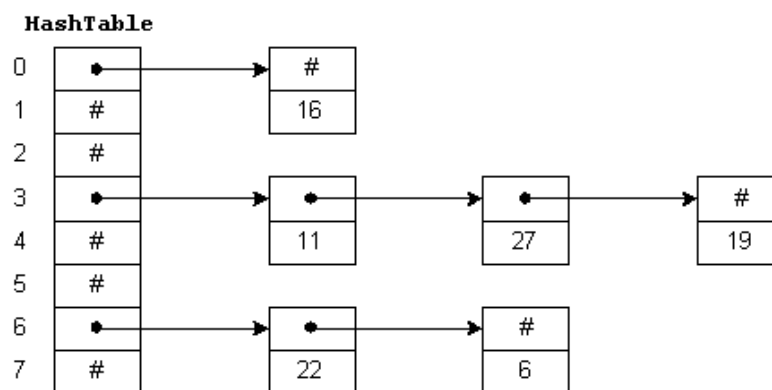
Referitor la prima condiție de mai sus, trebuie să observăm faptul că ea permite existența unor obiecte diferite din punct de vedere al conținutului, dar care au asociate aceeași valoare de dispersie (i.e., dacă `obiect_1 != obiect_2` atunci este posibil ca `hash(obiect_1) == hash(obiect_2)`)! În acest caz, spunem că funcția de dispersie are *coliziuni*. O funcție de dispersie ideală ar trebui să nu aibă coliziuni, dar practic acest lucru este imposibil din cauza *principiului lui Dirichlet*: "*Indiferent de modul în care vom plasa $n + 1$ obiecte în n cutii, va exista cel puțin o cutie care va conține două obiecte*". Astfel, considerând faptul că o funcție de dispersie furnizează valori de dispersie pe 32 biți, rezultă că acestea vor fi numere întregi cuprinse între -2.147.483.648 și 2.147.483.647, deci vor exista 4.294.967.294 valori distincte posibile pentru valorile de dispersie generate de funcția respectivă, ceea ce înseamnă că după aplicarea funcției de dispersie asupra a 4.294.967.294 + 1 obiecte distincte se va obține sigur cel puțin o coliziune!

Referitor la a doua condiție de mai sus, trebuie să observăm faptul că ea restricționează implementarea unei funcții de dispersie doar pentru obiecte imutabile (i.e., al căror conținut nu mai poate fi modificat după ce au fost create)!

Folosind funcții de dispersie se pot crea *tabele de dispersie (hash tables)*, care sunt structuri de date foarte eficiente din punct de vedere al operațiilor de inserare, căutare și ștergere, acestea având, în general, o complexitate computațională medie constantă.

Practic, o tabelă de dispersie este o structură de date asociativă în care unui obiect i se asociază un index (numit și *cheia obiectului*) într-un tablou unidimensional, calculat pe baza valorii de dispersie asociată obiectului respectiv prin intermediul unei funcții de dispersie.

În cazul unei funcții de dispersie ideale (i.e., care nu are coliziuni), fiecărui index din tablou îi va corespunde un singur obiect, dar, în realitate, din cauza, coliziunilor, vor exista mai multe obiecte asociate aceluiași index, care vor fi memorate folosind diverse structuri de date (e.g., liste simplu sau dublu înlănțuite). De exemplu, să considerăm numerele 16, 11, 27, 22, 19 și 6, precum și funcția de dispersie $h(x) = x \% 8$ corespunzătoare unei tabele de dispersie cu 8 elemente. Pe baza valorilor de dispersie, cele 6 numere vor fi distribuite în tabelă de dispersie astfel (sursa imaginii: <https://howtodoinjava.com/java/collections/hashtable-class/>):



Observați faptul că valorile dintr-o listă sunt, de fapt, coliziuni ale funcției de dispersie: $h(11) = h(27) = h(19) = 3$!

Încheiem prin a preciza faptul că performanțele unei tabele de dispersie depind de mai mulți factori (e.g., dimensiunea tablei, funcția de dispersie utilizată, modalitatea de rezolvare a coliziunilor etc.), dar, în general, operațiile de inserare, căutare și ștergere se vor efectua cu complexitatea medie $O(1)$. Mai multe detalii referitoare la tabelele de dispersie puteți să găsiți aici: https://itlectures.ro/wp-content/uploads/2020/04/SDA_curs6_hashTables_new.pdf.

Mulțimi

O *mulțime* este o colecție mutabilă de valori fără duplicate. Valorile memorate într-o mulțime pot fi neomogene (i.e., pot fi de tipuri diferite de date), dar trebuie să fie imutabile, deoarece mulțimile sunt implementate intern folosind tabele de dispersie. Mulțimile nu sunt indexate și nu păstrează ordinea de inserare a elementelor. Mulțimile sunt instanțe ale clasei `set`.

O mulțime poate fi creată/inițializată în mai multe moduri:

- folosind un șir de constante sau o secvență:

```
# multime vidă
s = set()
print(s)          # set()

# șir de constante numerice
s = {3, 2, 1, 1, 2, 3, 3, 4, 1}
print(s)          # {1, 2, 3, 4}

# listă de numere
s = set([4, 3, 2, 1, 4, 4, 3, 7, 1])
print(s)          # {1, 2, 3, 4, 7}

# șir de caractere
s = set("testare")
print(s)          # {'s', 'a', 't', 'r', 'e'}
```

- folosind secvențe de inițializare:

```
# resturi distincte
s = {x % 2 for x in range(100)}
print(s)          # {0, 1}

# prenume distincte
lista = ["Ana", "IoN", "ANA", "ana", "George", "IoN", "ION"]
s = set(nume.upper() for nume in lista)
print(s)          # {'GEORGE', 'ANA', 'ION'}

# grupe distincte
lista = [("Popa Anca", 134), ("Popescu Ion", 131),
         ("Ion Mihai", 134), ("Georgescu Ana", 133),
         ("Radu Ileana", 131), ("Pop Ion", 131)]
s = set(t[1] for t in lista)
print(s)          # {131, 133, 134}
```

Accesarea elementelor unei mulțimi

Deoarece elementele unei mulțimi nu sunt indexate, acestea pot fi accesate doar printr-o parcurgere secvențială:

```
s = {1, 2, 3, 2, 2, 4, 1, 1, 7}
for x in s:
    print(x, end=" ")  # 1 2 3 4 7
```

Operatori pentru mulțimi

În limbajul Python sunt definiți următorii operatori pentru manipularea mulțimilor:

a) *operatorii pentru testarea apartenenței*: `in`, `not in`

Exemplu: expresia `3 in {2, 3, 4, 3, 5, 2}` va avea valoarea `True`

b) *operatorii relaționali*: `<`, `<=`, `>`, `>=`, `==`, `!=`

Observații:

- În cazul primilor 4 operatori, cele două mulțimi vor fi comparate din punct de vedere al relației matematice de incluziune (submulțime / supermulțime)!
- În cazul ultimilor 2 operatori, nu se va ține cont de ordinea elementelor din cele două mulțimi comparate, ci doar de valorile lor!

Exemple:

```
S1 = {1, 2, 3, 4, 5, 100}
S2 = {1, 2, 4}
print(S2 < S1)           # True
print(S1 >= S2)          # True

S3 = {4, 1, 2, 1, 4, 4}
print(S3 < S2)           # False
print(S3 <= S2)          # True
print(S3 == S2)          # True
```

c) *Operatorii pentru operații specifice mulțimilor*: `|` (reuniune), `&` (intersecție), `-` (diferență), `^` (diferență simetrică, i.e. $A \triangle B = (A \setminus B) \cup (B \setminus A)$)

Exemple:

```
A = {1, 3, 4, 7}
B = {1, 2, 3, 4, 6, 8}

print("Reuniunea:", A | B)           # {1, 2, 3, 4, 6, 7, 8}
print("Intersecția:", A & B)          # {1, 3, 4}
print("Diferența A\B:", A - B)       # {7}
print("Diferența B\A:", B - A)       # {2, 6}
print("Diferența simetrică:", A ^ B) # {2, 6, 7, 8}
```

Funcții predefinite pentru mulțimi

În limbajul Python sunt predefinite mai multe funcții (*built-in functions* – <https://docs.python.org/3/library/functions.html>), dintre care unele pot fi utilizate pentru mai multe tipuri de date. De exemplu, funcția `len(secvență)` va furniza numărul de elemente dintr-o secvență (iterabil), indiferent dacă aceasta este o listă, un tuplu, un șir de caractere sau o mulțime.

Funcțiile predefinite care se pot utiliza pentru mulțimi sunt următoarele:

- a) **len(mulțime)**: furnizează numărul de elemente din mulțime (cardinalul mulțimii)

Exemplu: `len({2, 1, 3, 3, 2, 1, 7, 3}) = 4`

- b) **set(secvență)**: furnizează o mulțime formată din elementele secvenței respective

Exemplu: `set("teste") = {'e', 't', 's'}`

- c) **min(mulțime) / max(mulțime)**: furnizează elementul minim/maxim în sens lexicografic din mulțimea respectivă (atenție, toate elementele mulțimii trebuie să fie comparabile între ele, altfel va fi generată o eroare!)

Exemple:

```
S = {100, -70, 16, 100, -85, 100, -70, 28}
print("Minimul din mulțime:", min(S))      # -85
print("Maximul din mulțime:", max(S))      # 101
print()

S = {(2, 10), (2, 1, 2), (60, 2, 1), (3, 140, 5)}
print("Minimul din mulțime:", min(S))      # (2, 1, 2)
print("Maximul din mulțime:", max(S))      # (60, 2, 1)

S = {20, -30, "101", 17, 100}
print("Minimul din mulțime:", min(S))
# TypeError: '<' not supported between
# instances of 'str' and 'int'
```

- d) **sum(mulțime)**: furnizează suma elementelor unei mulțimi (evident, toate elementele mulțimii trebuie să fie de tip numeric)

Exemplu: `sum({1, -7, 10, -8, 10, -7}) = -4`

- e) **sorted(mulțime, [reverse=False])**: furnizează o listă formată din elementele mulțimii respective sortate crescător (mulțimea nu va fi modificată!).

Exemplu: `sorted({1, -7, 1, -8, 1, -7}) = [-8, -7, 1]`

Elementele mulțimii pot fi sortate și descrescător, setând parametrul opțional `reverse` al funcției `sorted` la valoarea `True`.

Exemplu: `sorted({1, -7, 1, -8}, reverse=True) = [1, -7, -8]`

Metode pentru prelucrarea mulțimilor

Metodele pentru prelucrarea mulțimilor sunt, de fapt, metodele încapsulate în clasa `set`. Așa cum am precizat anterior, mulțimile sunt *mutabile*, deci metodele respective pot modifica mulțimea curentă, dacă acest lucru este necesar!

În continuare, vom prezenta mai multe metode pentru prelucrarea mulțimilor, cu observația că parametrii scriși între paranteze drepte sunt opționali:

a) **`add(valoare)`**: dacă valoarea nu există deja în mulțime, atunci o adaugă.

Exemplu:

```
S = {1, 3, 5, 7, 9}
print(S)      # {1, 3, 5, 7, 9}

S.add(4)
print(S)      # {1, 3, 4, 5, 7, 9}

S.add(4)
print(S)      # {1, 3, 4, 5, 7, 9}
```

b) **`update(secvență)`**: adaugă, pe rând, toate elementele din secvența dată în mulțime.

Exemplu:

```
S = {1, 2, 3}
S.add((2, 3, 4, 5, 4, 5))
print(S)      # {1, 2, 3, (2, 3, 4, 5, 4, 5)}
```

```
S = {1, 2, 3}
S.update((2, 3, 4, 5, 4, 5))
print(S)      # {1, 2, 3, 4, 5}
```

```
S = {1, 2, 3}
S.update([4, 2, 3, 4, (4, 5)])
print(S)      # {1, 2, 3, 4, (4, 5)}
```

c) **`remove(valoare)`**: șterge din mulțimea curentă valoarea indicată sau lansează o eroare (`KeyError`) dacă valoarea nu apare în mulțime.

Exemple:

Pentru a evita apariția erorii `KeyError`, mai întâi am verificat faptul că valoarea `x` pe care dorim să o ștergem se găsește în mulțime:

```
S = {3, 2, 1, 1, 2, 3, 3}
print(f"Multimea: {S}")

x = 30
if x in S:
    S.remove(x)
print(f"Multimea dupa stergerea valorii {x}: {S}!")
```

```
else:
    print(f"Valoarea {x} nu apare in multime!")
```

O altă modalitate de utilizare a metodei `remove`, mai eficientă, constă în tratarea erorii care poate să apară când valoarea `x` pe care dorim să o ștergem nu se găsește în mulțime:

```
S = {3, 2, 1, 1, 2, 3, 3}
print(f"Multimea: {S}")

x = 30
try:
    S.remove(x)
    print(f"Multimea dupa stergerea valorii {x}: {S}!")
except KeyError:
    print(f"Valoarea {x} nu apare in multime!")
```

- d) **`discard(valoare)`**: șterge din mulțimea curentă valoarea indicată, dacă aceasta se găsește în mulțime, altfel mulțimea nu va fi modificată.

Exemplu:

```
S = {3, 2, 1, 1, 2, 3, 3}
print(f"Multimea: {S}")

x = 30

S.discard(x)
print(f"Multimea dupa stergerea valorii {x}: {S}!")
```

- e) **`clear()`**: șterge toate elementele din mulțime.

Exemplu:

```
S = {3, 2, 1, 1, 2, 3, 3}
print(f"Multimea: {S}")      # Multimea: {1, 2, 3}

S.clear()
print(f"Multimea: {S}")      # Multimea: set()
```

- f) **`union(secvență)`, `intersection(secvență)`, `difference(secvență)`, `symmetric_difference(secvență)`**: furnizează mulțimea obținută prin aplicarea operației respective asupra mulțimii curente și secvenței respective (convertită automat în mulțime), fără a modifica mulțime curentă!

Exemplu:

```
A = {1, 3, 4, 7}
B = [1, 2, 3, 4, 6, 8]

print("Reuniunea:", A.union(B))      # {1, 2, 3, 4, 6, 7, 8}
print("Intersecția:", A.intersection(B))  # {1, 3, 4}
```



```
print("Diferența A\B:", A.difference(B))      # {7}
print("Diferența simetrică:", A.symmetric_difference(B))
                                                # {2, 6, 7, 8}
print("Multimea A: ", A)      # {1, 3, 4, 7}
print("Lista B:", B)          # [1, 2, 3, 4, 6, 8]
```

- g) **intersection_update(secvență), difference_update(secvență), symmetric_difference_update(secvență):** înlocuiește mulțimea curentă cu mulțimea obținută prin aplicarea operației respective asupra mulțimii curente și secvenței respective (convertită automat în mulțime)!

Exemplu:

```
A = {1, 3, 4, 7}
B = [1, 2, 3, 4, 6, 8]

A.intersection_update(B)
print("Multimea A:", A)      # {1, 3, 4}

A.update([7, 5, 7, 4, 3])
print("Multimea A:", A)      # {1, 3, 4, 5, 7}

A.difference_update(B)
print("Multimea A:", A)      # {5, 7}

A.symmetric_difference_update(B)
print("Multimea A:", A)      # {1, 5, 2, 7, 3, 4, 6, 8}
```

Crearea unei mulțimi

O mulțime poate fi creată folosind valori constante, secvențe de valori, valori citite de la tastatură sau valori citite dintr-un fișier. Indiferent de sursa elementelor utilizate pentru crearea mulțimii, există mai multe variante de implementare pe care le putem utiliza: secvențe de inițializare, adăugarea unui element folosind metoda `add` sau operatorul `+=` sau reunirea la mulțimea curentă a unei mulțimi formate doar din elementul pe care dorim să-l adăugăm. În continuare, vom testa toate aceste variante din punct de vedere al timpului de executare, creând, de fiecare dată, o mulțime formată cu 200000 de elemente, respectiv numerele 0, 1, 2, ..., 199999:

```
import time

nr_elemente = 200000

start = time.time()
multime = set(x for x in range(nr_elemente))
stop = time.time()
print("    Initializare: ", stop - start, "secunde")
```

```

start = time.time()
multime = set()
for x in range(nr_elemente):
    multime.add(x)
stop = time.time()
print("    Metoda add(): ", stop - start, "secunde")

start = time.time()
multime = set()
for x in range(nr_elemente):
    multime |= {x}
stop = time.time()
print("    Operatorul |=: ", stop - start, "secunde")

start = time.time()
multime = set()
for x in range(nr_elemente):
    multime = multime | {x}
stop = time.time()
print("    Operatorul |: ", stop - start, "secunde")

```

Rezultatele obținute sunt următoarele:

```

Initializare: 0.015614748001098633 secunde
Metoda add(): 0.015626907348632812 secunde
Operatorul |=: 0.03124070167541504 secunde
Operatorul |: 408.1307489871979 secunde

```

Se observă faptul că primele 3 variante au timpi de executare mici, aproximativ egali, iar ultima variantă are un timp de executare mult mai mare din cauza faptului că la fiecare operație de reunire a mulțimii {x} la mulțimea curentă se creează în memorie o copie a mulțimii curente, se reunește la copie noua valoare x și apoi referința mulțimii curente se înlocuiește cu referința copiei.

Dicționare

Un *dicționar* este o colecție de date asociativă (*tablou asociativ*), deci permite asocierea unei valori arbitrare cu o cheie unică. Astfel, accesarea unei valori dintr-un dicționar nu se realizează prin poziția sa în tablou (index), ci prin cheia sa. Practic, putem privi un dicționar ca fiind o colecție de perechi de forma *cheie: valoare*. Cheile unui dicționar trebuie să fie unice și imutabile, dar pentru valori nu există nicio restricție. Începând cu versiunea Python 3.7, dicționarele păstrează ordinea de inserare a cheilor. Dicționarele sunt instanțe ale clasei `dict`.

Un dicționar poate fi creat/inițializat în mai multe moduri:

- folosind constante, operații de inserare sau funcția `dict`:

```
# dicționar vid - {}
d = {}
print(d)

# dicționar cu chei neomogene, dar imutabile
d = {"A": 4, "B": "Popa Ion", 6: -100,
      (1, 2, 3): [100, 200, 300]}
print(d)

# inserarea unor perechi cheie: valoare într-un dicționar
d = {}
d["A"] = 4
d["B"] = "Popa Ion"
d[6] = -100
d[(1, 2, 3)] = [100, 200, 300]
print(d)

# preluarea perechilor cheie: valoare dintr-o listă
# de tuple-uri de forma (cheie, valoare)
d = dict([("A", 4), ("B", "Popa Ion"), (6, -100),
          ((1, 2, 3), [100, 200, 300])])
print(d)
```

- folosind secvențe de inițializare:

```
# dicționar cu perechi literă: cod ASCII
d = {chr(65 + x): 65+x for x in range(5)}
print(d)      # {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E': 69}

# dicționar cu frecvențele literelor unui cuvânt,
# respectiv perechi literă: frecvență
cuv = "testare"
d = {lit: cuv.count(lit) for lit in set(cuv)}
print(d)      # {'a': 1, 't': 2, 's': 1, 'e': 2, 'r': 1}

# dicționar cu perechi număr: suma cifrelor
lista = [2134, 456, 777, 8144, 9]
d = {x: sum([int(c) for c in str(x)]) for x in lista}
print(d)      # {2134: 10, 456: 15, 777: 21, 8144: 17, 9: 9}
```

Accesarea elementelor unui dicționar

Elementele unui dicționar sunt indexate prin cheile asociate, deci pot fi accesate doar prin intermediul cheilor respective, ci nu prin pozițiile lor:

```
# dicționar cu perechi literă: cod ASCII
d = {chr(65 + x): x+65 for x in range(5)}
print(d)          # {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E': 69}

# modificare valorii asociate unei chei (i.e., cheia "B")
d["B"] = 10
print(d)          # {'A': 65, 'B': 10, 'C': 67, 'D': 68, 'E': 69}

# ștergere unei chei (i.e., cheia "B")
del d["B"]
print(d)          # {'A': 65, 'C': 67, 'D': 68, 'E': 69}

# inserarea unei chei noi și a unei valori asociate
# (i.e., noii chei "K" i se asociază valoarea 7)
d["K"] = 7
print(d)          # {'A': 65, 'C': 67, 'D': 68, 'E': 69, 'K': 7}
```

Dacă într-un dicționar se încearcă accesarea unei chei inexistente, atunci va fi lansată o eroare de tipul `KeyError`:

```
# dicționar cu perechi literă: cod ASCII
d = {chr(65 + x): x+65 for x in range(5)}
print(d)          # {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E': 69}

k = "Z"
print(d[k])       # KeyError: 'Z'
```

Pentru a evita apariția erorii precizate anterior, fie putem să testăm mai întâi existența cheii respective în dicționar, fie să tratăm excepția `KeyError`:

```
# dicționar cu perechi literă: cod ASCII
# d = {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E': 69}
d = {chr(65 + x): x+65 for x in range(5)}

k = "A"
if k in d:
    print(f"d['{k}'] = {d[k]}")
else:
    print(f"Cheia {k} nu apare în dicționar!")

k = "Z"
try:
    print(f"d['{k}'] = {d[k]}")
except KeyError:
    print(f"Cheia {k} nu apare în dicționar!")
```

Dicționarele pot fi utilizate pentru a organiza eficient anumite informații din punct de vedere al complexității computaționale a operațiilor de inserare, accesare sau ștergere (i.e., o complexitate medie egală cu $\mathcal{O}(1)$). Astfel, printr-o alegere judicioasă a cheilor și valorilor asociate lor, se pot optimiza procesele de actualizare sau interogare a unor date.

De exemplu, să considerăm următoarele informații despre $n = 4$ studenți (numele, grupa și media), memorate într-o listă de tuple:

```
L = [("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]
```

Evident, aproape orice operație de actualizare sau interogare a acestor informații (e.g., afișarea sau modificarea informațiilor referitoare la un student, afișarea studenților dintr-o anumită grupă, afișarea studenților având o anumită medie etc.) se va realiza cu o complexitate maximă egală cu $\mathcal{O}(n)$, deoarece ea va necesita parcurgerea întregii liste.

Dacă într-un program vom efectua multe operații de actualizare sau interogare pe baza numelor studenților, atunci putem să organizăm informațiile într-un dicționar cu perechi de forma nume: (grupă, medie):

```
L = [("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]

dict_nume = {}
for t in L:
    dict_nume[t[0]] = (t[1], t[2])

print(f"Dictionar: {dict_nume}")

# dict_nume = {"Marinescu Ioana": (152, 9.85),
#              "Constantinescu Ion": (151, 7.70),
#              "Popescu Ion": (151, 9.70),
#              "Filip Anca": (152, 9.70)}

ns = "Popescu Ion"
print(f"{ns}: {dict_nume[ns]}")

ns = "Popescu Ion"
ms = 9.20
dict_nume[ns] = (dict_nume[ns][0], ms)
print(f"{ns}: {dict_nume[ns]}")

del dict_nume["Popescu Ion"]
print(f"Dictionar: {dict_nume}")
```

În acest caz, operațiile de actualizare și interogare se vor executa cu o complexitate medie mult mai bună, respectiv $\mathcal{O}(1)$. Atenție, această variantă de modelare este corectă doar în cazul în care nu există 2 studenți cu același nume! De ce?

Aceleași informații le putem organiza sub forma unui dicționar cu perechi de forma grupa: [lista studenților din grupă], dacă știm că vom efectua multe operații de actualizare sau interogare la nivel de grupă:

```
L = [("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]

dict_grupe = {}
for t in L:
    if t[1] not in dict_grupe:
        dict_grupe[t[1]] = [(t[0], t[2])]
    else:
        dict_grupe[t[1]].append((t[0], t[2]))

print(f"Dictionar: {dict_grupe}")

# dict_grupe = {152: [("Marinescu Ioana", 9.85),
#                    ("Filip Anca", 9.70)],
#               151: [("Constantinescu Ion", 7.70),
#                    ("Popescu Ion", 9.70)]}

print(dict_grupe[152])

dict_grupe[152].append(("Mihai Carmen", 8.85))
print(dict_grupe[152])
```

Cu toate acestea, operațiile de actualizare sau interogare a mediei unui student vor avea o complexitate maximă egală cu $\mathcal{O}(n)$, deoarece implică parcurgerea tuturor studenților din grupa studentului respectiv. În acest caz, putem să modificăm structura dicționarului, păstrând informațiile despre studenții dintr-o grupă nu într-o listă, ci într-un dicționar cu perechi de forma nume: medie:

```
L = [("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]

dict_grupe = {}
for t in L:
    if t[1] not in dict_grupe:
        dict_grupe[t[1]] = {t[0]: t[2]}
    else:
        dict_grupe[t[1]][t[0]] = t[2]
```

```
# dict_grupe = {152: {"Marinescu Ioana": 9.85,
#                  "Filip Anca": 9.70},
#              151: {"Constantinescu Ion": 7.70,
#                  "Popescu Ion": 9.70}}

print(dict_grupe[152]["Filip Anca"])

print(dict_grupe[152])

dict_grupe[152]["Mihai Carmen"] = 8.85
print(dict_grupe[152])
```

Astfel, operațiile de actualizare și interogare a notei unui student se vor executa cu o complexitate medie mult mai bună, respectiv $O(1)$. Atenție, această variantă de modelare este corectă doar în cazul în care nu există 2 studenți cu același nume în aceeași grupă!

Dacă în programul nostru vom efectua multe operații de actualizare sau interogare în funcție de mediile studenților (de exemplu, pentru cazare sau acordarea burselor), atunci vom organiza informațiile sub forma unui dicționar cu perechi de forma medie: [lista cu tupluri (nume, grupa)]:

```
L = [("Marinescu Ioana", 152, 9.85),
     ("Constantinescu Ion", 151, 7.70),
     ("Popescu Ion", 151, 9.70),
     ("Filip Anca", 152, 9.70)]

dict_medii = {}
for t in L:
    if t[2] not in dict_medii:
        dict_medii[t[2]] = [(t[0], t[1])]
    else:
        dict_medii[t[2]].append((t[0], t[1]))

# dict_medii = {9.85: [("Marinescu Ioana", 152)],
#              9.70: [("Filip Anca", 152), ("Popescu Ion",
#              151)],
#              7.70: [("Constantinescu Ion", 152)]}

m = 9.70
print(f"Studentii cu media {m}: {dict_medii[m]}")
```

Operatori pentru dicționare

În limbajul Python sunt definiți următorii operatori pentru manipularea dicționarelor:

a) *operatorii pentru testarea apartenenței la nivel de cheie*: in, not in

Exemplu: expresia 'B' in {'A': 10, 'B': 7, 'C': 4, 'D': 7} va avea valoarea True, iar 7 in {'A': 10, 'B': 7, 'C': 4, 'D': 7} va avea False

- b) *operatorii relaționali*: `==`, `!=` (două dicționare sunt egale dacă sunt formate din aceleași perechi cheie: valoare, indiferent de ordinea de inserare)

Exemple:

```
d1 = {'A': 10, 'B': 7, 'C': 4, 'D': 7}
d2 = {'D': 7, 'A': 10, 'C': 4, 'B': 7}
d3 = {'A': 10, 'B': 17, 'C': 4, 'D': 7}
print(d1 == d2)      # True
print(d1 == d3)      # False
print(d2 != d3)      # True
```

Funcții predefinite pentru dicționare

În limbajul Python sunt predefinite mai multe funcții (*built-in functions* – <https://docs.python.org/3/library/functions.html>), dintre care unele pot fi utilizate pentru mai multe tipuri de date. De exemplu, funcția `len` furnizează numărul de elemente dintr-o secvență (iterabil), indiferent dacă aceasta este o listă, un tuplu, un șir de caractere sau o mulțime, dar și numărul cheilor dintr-un dicționar.

Funcțiile predefinite care se pot utiliza pentru dicționare sunt următoarele:

- a) **`len(dicționar)`**: furnizează numărul cheilor dicționarului

Exemplu: `len({'A': 10, 'B': 7, 'C': 4, 'D': 7}) = 4`

- b) **`dict(secvență)`**: furnizează un dicționar format din elementele secvenței respective, care trebuie să fie toate perechi (primul element al unei perechi este considerat o cheie, iar al doilea reprezintă valoarea asociată cheii respective)

Exemplu: `dict([('A',10), ('B',7), ('C',4), ('D',7)]) = {'A': 10, 'B': 7, 'C': 4, 'D': 7}`

- c) **`min(dicționar)` / `max(dicționar)`**: furnizează elementul minim/maxim în sens lexicografic din mulțimea respectivă (atenție, toate elementele mulțimii trebuie să fie comparabile între ele, altfel va fi generată o eroare!)

Exemple:

```
d = {'E':20, 'D': 7, 'A': 10, 'C': 40, 'B': 7}
print(f"Cheia minima: '{min(d)}'")      # Cheia minima: 'A'
print(f"Cheia maxima: '{max(d)}'")      # Cheia maxima: 'E'
```

- d) **`sum(dicționar)`**: furnizează suma cheilor unui dicționar (evident, toate cheile trebuie să fie de tip numeric)

Exemplu: `sum({20: 'E', 7: 'B', 10: 'A', 40: 'C'}) = 77`

- e) **sorted(dicționar, [reverse=False]):** furnizează o listă formată din cheile dicționarului respectiv sortate crescător (dicționarul nu va fi modificat!).

Exemplu: `sorted({20: 'E', 7: 'B', 40: 'C'}) = [7, 20, 40]`

Elementele mulțimii pot fi sortate și descrescător, setând parametrul opțional `reverse` al funcției `sorted` la valoarea `True`.

Exemplu: `sorted({'E': 20, 'B': 7, 'C': 40}, reverse=True) = ['E', 'C', 'B']`

Metode pentru prelucrarea dicționarelor

Metodele pentru prelucrarea dicționarelor sunt, de fapt, metodele încapsulate în clasa `dict`. Așa cum am precizat anterior, dicționarele sunt *mutabile*, deci metodele respective pot modifica dicționarul curent, dacă acest lucru este necesar!

În continuare, vom prezenta mai multe metode pentru prelucrarea dicționarelor, cu observația că parametrii scriși între paranteze drepte sunt opționali:

- a) **get(cheie, [valoare eroare]):** furnizează valoarea asociată cheii respective dacă aceasta există în dicționar sau `None` în caz contrar. Dacă se precizează pentru parametrul opțional `valoare eroare` o anumită valoare, aceasta va fi furnizată în cazul în care cheia nu există în dicționar.

Exemplu:

```
d = {'E':20, 'D': 7, 'A': 10, 'C': 40, 'B': 7}

print(d.get('A'))           # 10
print(d.get('Z'))           # None
print(d.get('Z', -1000))    # -1000
```

În general, se recomandă utilizarea metodei `get` în locul accesării directe a unei chei dintr-un dicționar pentru a evita eroarea (de tip `KeyError`) care poate să apară dacă respectiva cheie nu se găsește în dicționar:

```
d = {'E':20, 'D': 7, 'A': 10, 'C': 40, 'B': 7}

if d.get('Z') == 100:      # NU
    print("DA")
else:
    print("NU")

if d['Z'] == 100:          # KeyError: 'Z'
    print("DA")
else:
    print("NU")
```

- b) **keys(), values(), items():** furnizează vizualizări (*dictionary views*) ale cheilor, valorilor sau perechilor (cheie, valoare) din dicționarul respectiv. Vizualizările sunt iterabile și vor fi actualizate dinamic în momentul modificării dicționarului (<https://docs.python.org/3/library/stdtypes.html#dict-views>). Vizualizările pot fi transformate în liste folosind funcția `list`.

Exemplu:

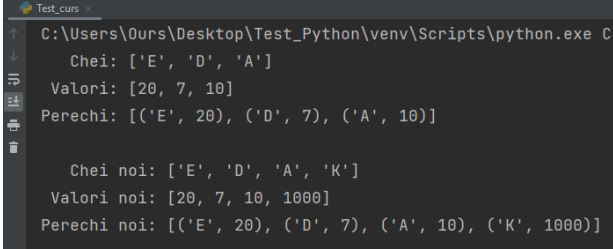
```
d = {'E':20, 'D': 7, 'A': 10}

k = d.keys()
v = d.values()
p = d.items()

print("  Chei:", list(k))
print(" Valori:", list(v))
print("Perechi:", list(p))

d['K'] = 1000

print()
print("  Chei noi:", list(k))
print(" Valori noi:", list(v))
print("Perechi noi:", list(p))
```



```
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
Chei: ['E', 'D', 'A']
Valori: [20, 7, 10]
Perechi: [('E', 20), ('D', 7), ('A', 10)]

Chei noi: ['E', 'D', 'A', 'K']
Valori noi: [20, 7, 10, 1000]
Perechi noi: [('E', 20), ('D', 7), ('A', 10), ('K', 1000)]
```

- c) **update(dicționar):** actualizează dicționarul curent folosind perechile cheie: valoare din dicționarul transmis ca parametru, astfel: dacă o cheie există deja în dicționarul curent atunci îi actualizează valoarea asociată, altfel adaugă o nouă cheie cu valoarea respectivă în dicționarul curent.

Exemplu:

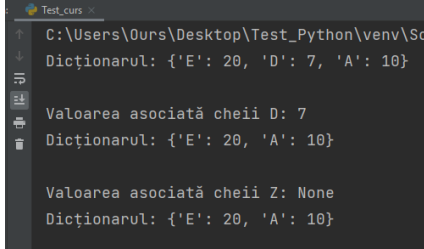
```
d = {'E':20, 'D': 7, 'A': 10}
t = {'D': -10, 'K': 5}

d.update(t)
print(d)          # {'E': 20, 'D': -10, 'A': 10, 'K': 5}
```

- d) **pop(cheie, [valoare implicită]):** șterge din mulțimea curentă cheia indicată, dacă aceasta există în dicționar, și furnizează valoarea asociată cu ea. Dacă cheia indicată nu se găsește în dicționar și parametrul opțional *valoare implicită* nu este setat, atunci va apărea o eroare, altfel metoda va furniza valoarea implicită setată.

Exemplu:

```
d = {'E':20, 'D': 7, 'A': 10}
print(f"Dicționarul: {d}\n")
k = 'D'
r = d.pop(k)
print(f"Valoarea asociată cheii {k}: {r}")
print(f"Dicționarul: {d}\n")
k = 'Z'
r = d.pop(k, None)
print(f"Valoarea asociată cheii {k}: {r}")
print(f"Dicționarul: {d}")
```



```
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
Dicționarul: {'E': 20, 'D': 7, 'A': 10}
Valoarea asociată cheii D: 7
Dicționarul: {'E': 20, 'A': 10}
Valoarea asociată cheii Z: None
Dicționarul: {'E': 20, 'A': 10}
```

- e) **clear():** șterge toate elementele din dicționar.

Complexitatea computațională a operațiilor asociate colecțiilor de date

O implementare optimă a unui algoritm presupune, de obicei, și utilizarea unor structuri de date adecvate, astfel încât operațiile necesare să fie implementate cu o complexitate minimă. De exemplu, putem verifica dacă o valoare a fost deja utilizată într-un algoritm în mai multe moduri:

- memorăm toate valorile într-o listă sau un tuplu și testăm, folosind operatorul `in` sau metoda `count`, dacă valoarea respectivă se găsește într-o listă / un tuplu cu n elemente, deci vom avea o complexitate maximă egală cu $\mathcal{O}(n)$;
- dacă valorile sunt deja sortate, atunci putem să utilizăm căutarea binară pentru a testa dacă valoarea respectivă se găsește în lista / tuplul cu n elemente, deci vom avea o complexitate maximă egală cu $\mathcal{O}(\log_2 n)$;
- memorăm valorile într-o mulțime sau un dicționar și atunci putem să testăm existența valorii respective cu complexitate medie egală cu $\mathcal{O}(1)$.

În continuare, vom prezenta complexitatea computațională a operațiilor implementate de colecțiile din limbajul Python (<https://wiki.python.org/moin/TimeComplexity>):

a) *liste / tuple* (cu n elemente)

Operație / metodă / funcție	Complexitate maximă
Accesarea unui element prin index	$\mathcal{O}(1)$
Ștergerea unui element (instrucțiunea <code>del</code> și metoda <code>remove</code>)	$\mathcal{O}(n)$
Parcurgere	$\mathcal{O}(n)$
Căutarea unei valori (operatorii <code>in</code> și <code>not in</code> sau metoda <code>index</code>)	$\mathcal{O}(n)$
<code>len(listă sau tuplu)</code>	$\mathcal{O}(1)$
<code>append(valoare)</code>	$\mathcal{O}(1)$
<code>extend(secvență)</code>	$\mathcal{O}(\text{len}(\text{secvență}))$
<code>pop()</code>	$\mathcal{O}(1)$
<code>pop(poziție)</code>	$\mathcal{O}(n)$
<code>count(valoare)</code>	$\mathcal{O}(n)$
<code>insert(poziție, valoare)</code>	$\mathcal{O}(n)$
Extragerea unei secvențe	$\mathcal{O}(\text{len}(\text{secvență}))$
Ștergerea unei secvențe	$\mathcal{O}(n)$
Modificarea unei secvențe	$\mathcal{O}(\text{len}(\text{secvență}) + n)$
Sortare (funcția <code>sorted</code> și metoda <code>sort</code>)	$\mathcal{O}(n \log_2 n)$
Funcțiile <code>min</code> , <code>max</code> și <code>sum</code>	$\mathcal{O}(n)$
Copiere (metoda <code>copy</code>)	$\mathcal{O}(n)$
Multiplicare de k ori (operatorul <code>*</code>)	$\mathcal{O}(nk)$

b) *mulțimi* (cu n elemente)

Operație / metodă / funcție	Complexitate medie	Complexitate maximă
Testarea apartenenței unei valori (operatorii <code>in</code> și <code>not in</code>)	$O(1)$	$O(n)$
Adăugarea unui element (metoda <code>add</code>)	$O(1)$	$O(n)$
Ștergerea unui element (metodele <code>remove</code> și <code>discard</code>)	$O(1)$	$O(n)$
Creare	$O(n)$	$O(n^2)$
Parcurgere		$O(n)$
<code>len(mulțime)</code>		$O(1)$
<code>update(secvență)</code>	$O(\text{len}(\text{secvență}))$	$O(n * \text{len}(\text{secvență}))$
Reuniunea mulțimilor A și B (operatorul <code> </code> și metoda <code>union</code>)	$O(\text{len}(A) + \text{len}(B))$	
Intersecția mulțimilor A și B (operatorul <code>&</code> , metoda <code>intersection</code> și metoda <code>intersection_update</code>)	$O(\min(\text{len}(A), \text{len}(B)))$	$O(\text{len}(A) * \text{len}(B))$
Diferența mulțimilor A și B (operatorul <code>-</code> și metoda <code>difference</code>)	$O(\text{len}(A))$	
Diferența mulțimilor A și B (metoda <code>difference_update</code>)	$O(\text{len}(B))$	
Diferența simetrică a mulțimilor A și B (operatorul <code>^</code> și metoda <code>symmetric_difference</code>)	$O(\text{len}(A))$	$O(\text{len}(A) * \text{len}(B))$
Diferența simetrică a mulțimilor A și B (metoda <code>symmetric_difference_update</code>)	$O(\text{len}(B))$	$O(\text{len}(A) * \text{len}(B))$
Sortare (funcția <code>sorted</code>)		$O(n \log_2 n)$
Funcțiile <code>min</code> , <code>max</code> și <code>sum</code>		$O(n)$

c) *dicționare* (cu n chei)

Operație / metodă / funcție	Complexitate medie	Complexitate maximă
Testarea apartenenței unei chei (operatorii <code>in</code> și <code>not in</code>)	$O(1)$	$O(n)$
Accesarea unui element prin cheie	$O(1)$	$O(n)$
Ștergerea unui element (instrucțiunea <code>del</code> și metoda <code>pop</code>)	$O(1)$	$O(n)$
Creare	$O(n)$	$O(n^2)$
Parcurgere		$O(n)$

În continuare, vom determina, folosind diverse colecții de date, frecvențele cuvintelor dintr-o propoziție formată din n cuvinte și citită din fișierul text *text.txt*:

- a) determinăm mulțimea cuvintelor distincte din propoziție și apoi calculăm frecvența fiecărui cuvânt distinct:

```
f = open("text.txt")
prop = f.read()
f.close()

# înlocuim semnele de punctuație cu spații
for sep in ".,:;?!":
    prop = prop.replace(sep, " ")
cuvinte = prop.split()

# calculăm frecvențele cuvintelor distincte
for cuv in set(cuvinte):
    frcuv = cuvinte.count(cuv)
    print(f"Cuvantul {cuv} apare de {frcuv} ori")
```

Această variantă are complexitatea maximă $\mathcal{O}(n^2)$, deoarece pot exista maxim n cuvinte distincte în propoziție, iar metoda `count` are complexitatea $\mathcal{O}(n)$.

- b) calculăm frecvența fiecărui cuvânt distinct din propoziție folosind un dicționar cu perechi de forma *cuvânt: frecvență*:

```
f = open("text.txt")
prop = f.read()
f.close()

# înlocuim semnele de punctuație cu spații
for sep in ".,:;?!":
    prop = prop.replace(sep, " ")
cuvinte = prop.split()

# inițializăm dicționarul cu cuvintele distincte
# din propoziție, fiecare cuvânt cu frecvența 0
frcuv = {cuv: 0 for cuv in set(cuvinte)}

# calculăm frecvența fiecărui cuvânt distinct
for cuv in cuvinte:
    frcuv[cuv] += 1

# afișăm frecvențele cuvintelor distincte
for cuv in frcuv:
    print(f"Cuvantul {cuv} apare de {frcuv[cuv]} ori")
```

Această variantă de rezolvare are complexitatea maximă $\mathcal{O}(n)$, deoarece crearea și actualizarea dicționarului `frcuv` au, fiecare, complexitatea $\mathcal{O}(n)$.