

CURS 5

Colecții de date

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"Dicționar: {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_ume[ns]}")

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

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

```

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!)

Example:

```
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(dictionar)**: 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(dictionar, [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:

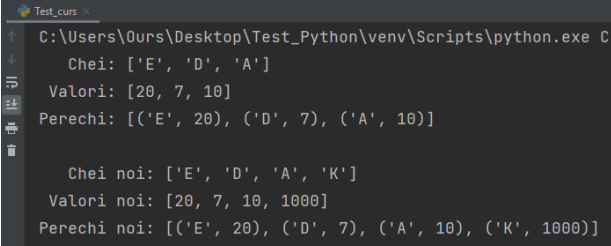
```
d = {'E':20, 'D': 7, 'A': 10}

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 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:

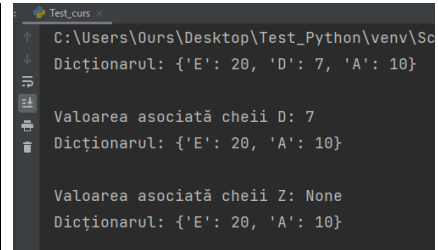
```
d = {'E':20, 'D': 7, 'A': 10}
t = {'D': -10, 'K': 5}

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"Dictionarul: {d}\n")
k = 'D'
r = d.pop(k)
print(f"Valoarea asociată cheii {k}: {r}")
print(f"Dictionarul: {d}\n")
k = 'Z'
r = d.pop(k, None)
print(f"Valoarea asociată cheii {k}: {r}")
print(f"Dictionarul: {d}")
```



```
Test_curs
C:\Users\Ours\Desktop\Test_Python\venv\Sc
Dictionarul: {'E': 20, 'D': 7, 'A': 10}

Valoarea asociată cheii D: 7
Dictionarul: {'E': 20, 'A': 10}

Valoarea asociată cheii Z: None
Dictionarul: {'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 $O(n)$;
- 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 $O(\log_2 n)$;
- memorăm valorile într-o mulțime sau un dicționar și atunci putem să testăm existența valorii respective cu complexitate medie egală cu $O(1)$.

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	$O(1)$
Ștergerea unui element (instrucțiunea <code>del</code> și metoda <code>remove</code>)	$O(n)$
Parcurgere	$O(n)$
Căutarea unei valori (operatorii <code>in</code> și <code>not in</code> sau metoda <code>index</code>)	$O(n)$
<code>len(listă sau tuplu)</code>	$O(1)$
<code>append(valoare)</code>	$O(1)$
<code>extend(secvență)</code>	$O(\text{len}(\text{secvență}))$
<code>pop()</code>	$O(1)$
<code>pop(poziție)</code>	$O(n)$
<code>count(valoare)</code>	$O(n)$

Operație / metodă / funcție	Complexitate maximă
insert(pozitie, valoare)	$O(n)$
Extragerea unei secvențe	$O(\text{len}(\text{secvență}))$
Ștergerea unei secvențe	$O(n)$
Modificarea unei secvențe	$O(\text{len}(\text{secvență}) + n)$
Sortare (funcția sorted și metoda sort)	$O(n \log_2 n)$
Funcțiile min, max și sum	$O(n)$
Copiere (metoda copy)	$O(n)$
Multiplicare de k ori (operatorul *)	$O(nk)$

b) *mulțimi* (cu n elemente)

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

c) *dictionare* (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.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 $O(n^2)$, deoarece pot exista maxim n cuvinte distincte în propoziție, iar metoda `count` are complexitatea $O(n)$.

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 $\mathcal{O}(n)$.