

Mape, heš tabele

Slajdovi sa predavanja¹

© Goodrich, Tamassia, Goldwasser

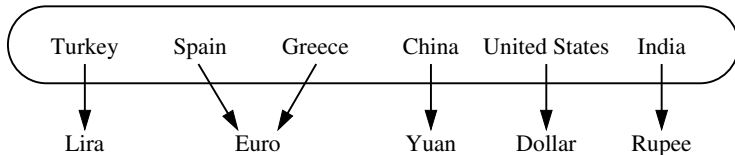
Katedra za informatiku, Fakultet tehničkih nauka, Univerzitet u Novom Sadu

2022.

¹Po uzoru na materijale sa: <https://github.com/mbranko/asp-slajdovi>

Mapa

- Pythonov rečnik (klasa **dict**) preslikava **ključeve** na **vrednosti**
- drugo ime: **asocijativni niz** ili **mapa**
- ključevi su jedinstveni (nema ponavljanja)
- vrednosti ne moraju biti jedinstvene



Mapa ATP: osnovne operacije

<code>M[k]</code>	vraća vrednost v vezanu za ključ k u mapi M ; ako ne postoji, izaziva <code>KeyError</code> ; implementira je <code>__getitem__</code>
<code>M[k] = v</code>	dodeljuje vrednost v ključu k u mapi M ; ako ključ već postoji, zamenjuje staru vrednost; implementira je <code>__setitem__</code>
<code>del M[k]</code>	uklanja element sa ključem k iz mape M ; ako ne postoji, izaziva <code>KeyError</code> ; implementira je <code>_delitem__</code>
<code>len(M)</code>	vraća broj elemenata u mapi M ; implementira je <code>__len__</code>
<code>iter(M)</code>	generiše listu ključeva iz mape M ; implementira je <code>__iter__</code>

Mapa ATP: dodatne operacije

<code>k in M</code>	vraća <code>True</code> ako mapa M sadrži ključ k ; implementira je <code>__contains__</code>
<code>M.get(k, d=None)</code>	vraća $M[k]$ ako ključ k postoji u M ; inače vraća default vrednost d ; ne izaziva <code>KeyError</code>
<code>M.setdefault(k, d)</code>	ako k postoji u mapi, vraća $M[k]$; ako ne postoji, postavlja $M[k] = d$ i vraća d
<code>M.pop(k, d=None)</code>	uklanja element sa ključem k i vraća vezanu vrednost v ; ako ključ k nije u mapi M , vraća d ili izaziva <code>KeyError</code> ako je d jednako <code>None</code>

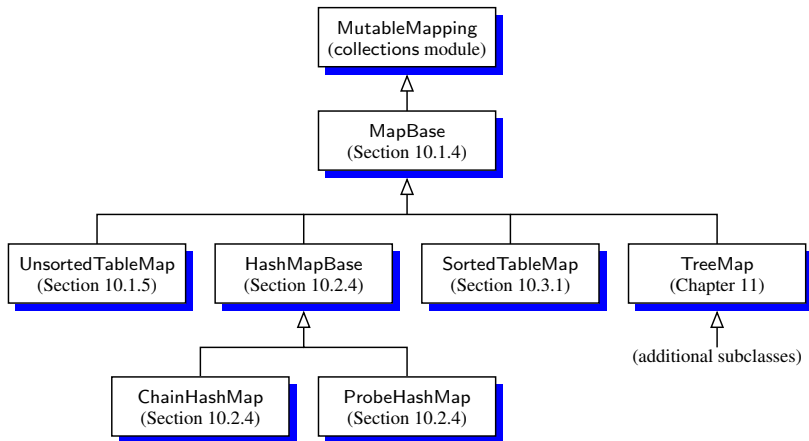
Mapa ATP: još malo operacija

M.popitem()	uklanja neki element mape i vraća (k,v) ; ako je mapa prazna izaziva <code>KeyError</code>
M.clear()	uklanja sve elemente iz mape
M.keys()	vraća skup svih ključeva iz M
M.values()	vraća skup svih vrednosti iz M
M.items()	vraća skup svih parova (k,v) iz M
M.update(M2)	dodeljuje $M[k]=v$ za svaki (k,v) iz $M2$
M == M2	vraća <code>True</code> ako mape sadrže iste parove (k,v)
M != M2	vraća <code>True</code> ako mape ne sadrže iste parove (k,v)

Mapa ATP: primer

operacija	rezultat	mapa
len(M)	0	{ }
M['K']=2	-	{'K':2}
M['B']=4	-	{'K':2, 'B':4}
M['U']=2	-	{'K':2, 'B':4, 'U':2}
M['V']=8	-	{'K':2, 'B':4, 'U':2, 'V':8}
M['K']=9	-	{'K':9, 'B':4, 'U':2, 'V':8}
M['B']	4	{'K':9, 'B':4, 'U':2, 'V':8}
M['X']	KeyError	{'K':9, 'B':4, 'U':2, 'V':8}
M.get('F')	None	{'K':9, 'B':4, 'U':2, 'V':8}
M.get('F', 5)	5	{'K':9, 'B':4, 'U':2, 'V':8}
M.get('K', 5)	9	{'K':9, 'B':4, 'U':2, 'V':8}
len(M)	4	{'K':9, 'B':4, 'U':2, 'V':8}
del M['V']	-	{'K':9, 'B':4, 'U':2}
M.pop('K')	9	{'B':4, 'U':2}
M.keys()	'B','U'	{'B':4, 'U':2}
M.values()	4,2	{'B':4, 'U':2}
M.items()	('B',4),('U',2)	{'B':4, 'U':2}
M.setdefault('B',1)	4	{'B':4, 'U':2}
M.setdefault('A',1)	1	{'A':1, 'B':4, 'U':2}
M.popitem()	('B',4)	{'A':1, 'U':2}

Različite implementacije mape

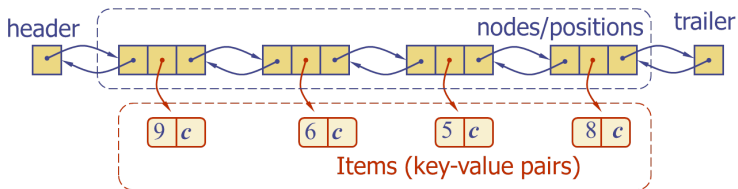


Implementacija MapElement

```
class MapElement(object):  
    """Klasa modeluje element asocijativnog niza."""  
  
    def __init__(self, k, v):  
        self._key = k  
        self._value = v  
  
    @property  
    def key(self):  
        return self._key  
  
    @property  
    def value(self):  
        return self._value  
  
    @value.setter  
    def value(self, new_value):  
        self._value = new_value
```


Mapa pomoću liste

- jedna moguća implementacija mape je pomoću dvostruko spregnute liste
- elemente čuvamo u proizvoljnom redosledu



Mapa pomoću liste: implementacija₁

```
class Map(object):
    """Klasa modeluje asocijativni niz."""

    def __init__(self):
        self._data = []

    def __getitem__(self, key):
        """
        Pristup elementu sa zadatim ključem

        Metoda vrši pristup elementu sa zadatim ključem. U slučaju
        da element postoji u mapi, metoda vraća njegovu vrednost, dok
        u suprotnom podiže odgovarajući izuzetak.

        Argument:
        - `key`: ključ elementa kome se pristupa
        """

        for item in self._data:
            if key == item.key:
                return item.value
        raise KeyError('Ne postoji element sa ključem %s' % str(key))
```

Mapa pomoću liste: implementacija₂

```
def __setitem__(self, key, value):
    """
    Postavljanje vrednosti elementa sa zadatim ključem

    Metoda najpre pretražuje postojeće elemente po vrednosti ključa.
    Ukoliko traženi ključ već postoji, vrši se ažuriranje vrednosti
    postojećeg elementa. U suprotnom, kreira se novi element koji se
    dodaje u mapu.

    Argumenti:
    - `key`: ključ elementa koji se kreira ili ažurira
    - `value`: nova vrednost elementa
    """
    for item in self._data:
        if key == item.key:
            item.value = value
            return

    # element nije pronađen, zapiši ga u mapu
    self._data.append(MapElement(key, value))
```

Mapa pomoću liste: implementacija₃

```
def __delitem__(self, key):
    """
    Brisanje elementa sa zadatim ključem

    Metoda pretražuje elemente po vrednosti ključa. Ukoliko element
    sa zadatim ključem postoji u mapi, vrši se njegovo brisanje. U
    suprotnom se podiže odgovarajući izuzetak.

    Argument:
    - `key`: ključ elementa za brisanje
    """
    length = len(self._data)
    for i in range(length):
        if key == self._data[i].key:
            self._data.pop(i)
            return

    raise KeyError('Ne postoji element sa ključem %s' % str(key))
```

Mapa pomoću liste: implementacija₄

```
def __len__(self):  
    return len(self._data)  
  
def __contains__(self, key):  
    """  
    Metoda vrši proveru postojanja ključa u mapi  
  
    Argument:  
    - `key`: ključ koji se traži  
    """  
    for item in self._data:  
        if key == item.key:  
            return True  
  
    return False  
  
def __iter__(self):  
    for item in self._data:  
        yield item.key
```

Mapa pomoću liste: implementacija₅

```
def items(self):
    for item in self._data:
        yield item.key, item.value

def keys(self):
    """
    Metoda vraća sve ključeve u mapi
    """
    keys = []
    for key in self:
        keys.append(key)

    return keys
```

Mapa pomoću liste: implementacija₆

```
def values(self):  
    """  
    Metoda vraća sve vrednosti u mapi  
    """  
    values = []  
    for key in self:  
        values.append(self[key])  
  
    return values  
  
def clear(self):  
    """  
    Metoda uklanja sve elemente iz mape  
    """  
    self._data = []
```

Mapa pomoću liste: performanse

- **dodavanje** traje $O(1)$ – novi element možemo dodati na početak ili na kraj
- **traženje** ili **uklanjanje** traje $O(n)$ – u najgorem slučaju (nije pronađen element) mora se proći kroz celu listu
- ovakva implementacija je korisna samo za mape sa malim brojem elemenata
- ili ako je dodavanje najčešća operacija, dok se traženje i uklanjanje retko obavljaju

Hash tabela

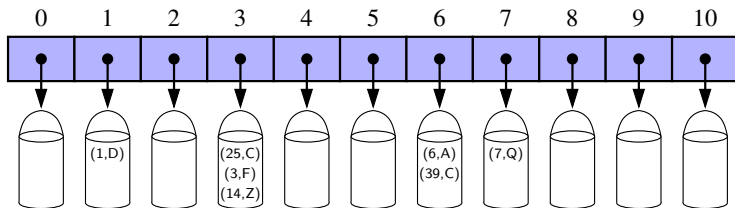
- mapa omogućava pristup korišćenjem **ključeva** kao **indeksa** – $M[k]$
- zamislimo mapu koja kao ključeve korisiti cele brojeve iz intervala $[0, N - 1]$ za neko $N > n$
- za čuvanje elemenata možemo koristiti **lookup** niz dužine N
- npr. mapa sa elementima $(1, D), (3, Z), (6, C), (7, Q)$

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

- operacije su $O(1)$

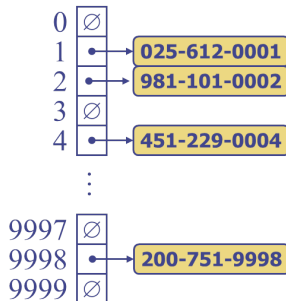
Hash tabela

- šta ako je $N \gg n$?
- šta ako ključevi nisu celi brojevi?
- pretvorićemo ključeve u cele brojeve pomoću **hash funkcije**
- dobra hash funkcija će ravnomerno distribuirati ključeve u $[0, N - 1]$
- ali može biti duplikata
- duplikate ćemo čuvati u „kantama“ – tzv. **bucket array**



Hash funkcija

- **hash funkcija** mapira ključeve na indekse u hash tabeli
- npr. poslednje četiri cifre broja telefona

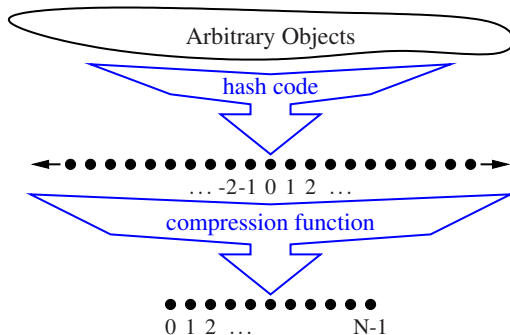


Hash funkcija

- **hash funkcija** mapira ključ k na ceo broj u intervalu $[0, N - 1]$
- gde je N kapacitet niza kanti A
- element (k, v) čuvamo u nizu kao $A[h(k)]$
- **kolizija**: dve vrednosti ključa koje daju isti hash
- dobre hash funkcije imaju **vrlo malo** kolizija

Hash funkcija

- često se hash funkcija može posmatrati kao kompozicija dve funkcije:
- **hash code**: mapira ključ na ceo broj
- **compression function**: mapira hash kôd na broj u intervalu $[0, N - 1]$



Hash funkcija

- ako se hash funkcija posmatra kao
hash code ◦ compression function
- tada hash code ne zavisi od veličine niza kanti
- vrednosti koje su „blizu“ u skupu ključeva ne moraju imati hasheve koji su „blizu“

Hash code ₁

- memorijska adresa
 - adresa Python objekta u memoriji kao hash code
 - dobro osim za numeričke tipove i stringove
- integer cast
 - za svaki tip podataka koji se predstavlja sa najviše onoliko bita koliko i **int** možemo uzeti int interpretaciju njegovih bita
 - za tipove koji zauzimaju više memorije moramo nekako „sažeti“ njegove bite
 - npr. **float** broj u Pythonu zauzima 64 bita a hash kod 32; možemo izabrati
 - gornjih 32 bita
 - donjih 32 bita
 - neku kombinaciju sva 64 bita: XOR ili zbir gornje i donje polovine, itd.

Hash code ₂

- suma komponenti
 - podelimo bitove ključa na delove po 32 bita
 - saberemo delove (ignorišemo overflow)
 - zgodno za numeričke ključeve duže od int-a
- polinomska akumulacija
 - podelimo bitove ključa na delove fiksne dužine (npr. 8, 16, 32 bita) $a_0 a_1 a_2 \dots a_{n-1}$
 - izračunamo polinom (ignorišući overflow) za fiksno z :

$$p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$$

Hash code ₃

- polinomska akumulacija
 - posebno zgodno za stringove ($z = 33$ daje samo 6 kolizija za 50.000 engleskih reči)
 - polinom $p(z)$ se može izračunati u $O(n)$ vremenu Hornerovom metodom
 - sledeći polinomi se računaju u $O(1)$ vremenu, svaki na osnovu prethodnog
 - $p_0(z) = a_{n-1}$
 - ...
 - $p_i(z) = a_{n-i-1} + zp_{i-1}(z)$
 - na kraju je: $p(z) = p_{n-1}(z)$

Kompresujuća funkcija

- celobrojno deljenje
 - $h(y) = y \bmod N$
 - veličina hash tabele N je obično prost broj
- multiply, add and divide (MAD)
 - $h(y) = (ay + b) \bmod N$
 - a i b su nenegativni celi brojevi takvi da je $a \bmod N \neq 0$

Apstraktna heš mapa₁

```
class HashMap(object):
    """
    Klasa modeluje heš mapu
    """

    def __init__(self, capacity=8):
        """
        Konstruktor

        Argumenti:
        - `capacity`: inicijalni broj mesta u lookup nizu
        - `prime`: prost broj neophodan heš funkciji
        """
        self._data = capacity * [None]
        self._capacity = capacity
        self._size = 0
        self.prime = 109345121

        # konstante heširanja
        self._a = 1 + random.randrange(self.prime-1)
        self._b = random.randrange(self.prime)

    def __len__(self):
        return self._size
```

Apstraktna heš mapa₂

```
def _hash(self, x):
    """
    Heš funkcija

    Izračunavanje heš koda vrši se po formuli  $(ax + b) \bmod p$ .

    Argument:
    - `x`: vrednost čiji se kod računa
    """
    hashed_value = (hash(x)*self._a + self._b) % self.prime
    compressed = hashed_value % self._capacity
    return compressed

def _resize(self, capacity):
    """
    Skaliranje broja raspoloživih slotova

    Metoda kreira niz sa unapred zadatim kapacitetom u koji
    se prepisuju vrednosti koje se trenutno nalaze u tabeli.

    Argument:
    - `capacity`: kapacitet novog niza
    """
    old_data = list(self.items())
    self._data = capacity * [None]
    self._size = 0

    # prepisivanje podataka u novu tabelu
    for (k, v) in old_data:
        self[k] = v
```

Apstraktna heš mapa₃

```
def __getitem__(self, key):
    """
    Pristup elementu sa zadatim ključem

    Apstraktna metoda koja opisuje pristup elementu na osnovu njegovog ključa. Implementacija pristupa bucketu varira u zavisnosti od načina rešavanja kolizija.

    Argument:
    - `key`: ključ elementa kome se pristupa
    """
    bucket_index = self._hash(key)
    return self._bucket_getitem(bucket_index, key)

def __setitem__(self, key, value):
    bucket_index = self._hash(key)
    self._bucket_setitem(bucket_index, key, value)

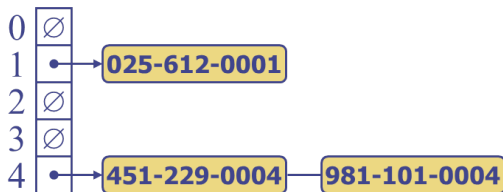
    # povećaj broj raspoloživih mesta
    current_capacity = len(self._data)
    if self._size > current_capacity // 2:
        self._resize(2*current_capacity - 1)
```

Apstraktna heš mapa₄

```
def __delitem__(self, key):  
    bucket_index = self._hash(key)  
    self._bucket_delitem(bucket_index, key)  
    self._size -= 1  
  
def items(self):  
    raise NotImplementedError()  
  
def _bucket_getitem(self, index, key):  
    raise NotImplementedError()  
  
def _bucket_setitem(self, index, key, value):  
    raise NotImplementedError()  
  
def _bucket_delitem(self, index, key):  
    raise NotImplementedError()
```

Rukovanje kolizijama

- kolizije nastaju kada se različiti elementi mapiraju na istu ćeliju
- ulančavanje duplikata: svaki element heš tabele je glava liste koja čuva elemente
- traži dodatnu memoriju pored same heš tabele



Heš mapa sa ulančavanjem₁

```
class ChainedHashMap(HashMap):
    """
    Klasa modeluje heš mapu koja kolizije rešava ulančavanjem
    """
    def _bucket_getitem(self, i, key):
        """
        Pristup elementu u okviru bucketa

        Metoda najpre pristupa bucketu sa zadatim indeksom. Ukoliko
        bucket postoji, pretražuje se. Ako je element pronađen metoda
        vraća njegovu vrednost, dok se u suprotnom podiže odgovarajući
        izuzetak.

        Argumenti:
        - `i`: indeks bucketa
        - `key`: ključ elementa
        """
        bucket = self._data[i]
        if bucket is None:
            raise KeyError('Ne postoji element sa traženim ključem.')

        return bucket[key]
```


Heš mapa sa ulančavanjem₂

```
def _bucket_setitem(self, bucket_index, key, value):
    """
    Postavljanje vrednosti elementa sa zadatim ključem

    Metoda najpre pronalazi bucket sa zadatim indeksom, a zatim
    dodaje novi element ili ažurira postojeći na osnovu zadatog
    ključa. Ukoliko bucket ne postoji, kreira se novi.

    Argumenti:
    - `i`: indeks bucketa
    - `key`: ključ elementa
    - `value`: vrednost elementa
    """
    bucket = self._data[bucket_index]
    if bucket is None:
        self._data[bucket_index] = Map()

    # broj elemenata u mapi se menja samo u slučaju da je došlo do
    # dodavanja, dok ažuriranje ne menja broj elemenata
    current_size = len(self._data[bucket_index])
    self._data[bucket_index][key] = value
    if len(self._data[bucket_index]) > current_size:
        self._size += 1
```

Heš mapa sa ulančavanjem₃

```
def _bucket_delitem(self, bucket_index, key):
    """
    Brisanje elementa sa zadatim ključem

    Metoda najpre pristupa bucketu sa zadatim indeksom. Ukoliko
    bucket postoji, pretražuje se. Ako je element pronađen vrši
    se njegovo brisanje, u suprotnom se podiže odgovarajući
    izuzetak.

    Argumenti:
    - `i`: indeks bucketa
    - `key`: ključ elementa
    """
    bucket = self._data[bucket_index]
    if bucket is None:
        raise KeyError('Ne postoji element sa traženim ključem.')

    del bucket[key]
```

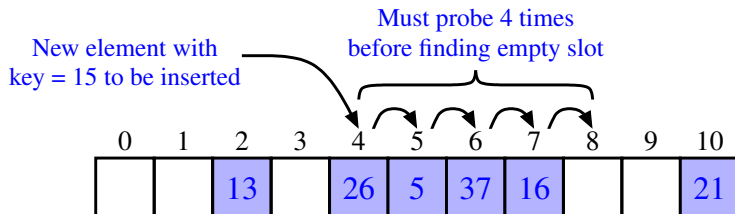
Heš mapa sa ulančavanjem₄

```
def __iter__(self):
    for bucket in self._data:
        if bucket is not None:
            for key in bucket:
                yield key

def items(self):
    for bucket in self._data:
        if bucket is not None:
            for key, value in bucket.items():
                yield key, value
```

Linearno traženje

- **linear probing**: smešta element u koliziji u prvu sledeću slobodnu ćeliju (cirkularno)
- elementi u koliziji se nagomilavaju izazivajući dalje kolizije
- primer:



Čitanje sa linearnim traženjem

- **get**(k): počinjemo od pozicije $h(k)$
- ispitujemo sledeće lokacije sve dok se ne dogodi nešto od:
 - pronašli smo ključ k
 - naišli smo na praznu lokaciju
 - ispitali smo N lokacija

get(k)

```

 $i \leftarrow h(k)$ 
 $p \leftarrow 0$ 
repeat
   $c \leftarrow A[i]$ 
  if  $c = \emptyset$  then
    return null
  else
    if  $c.getKey() = k$  then
      return  $c.getValue()$ 
    else
       $i \leftarrow (i + 1) \bmod N$ 
       $p \leftarrow p + 1$ 
until  $p = N$ 
return null
  
```

Izmene sa linearnim traženjem

- uvodimo poseban objekat
AVAILABLE koji zamenjuje uklonjene elemente
- **remove**(k)
 - tražimo element sa ključem k
 - ako smo ga našli, vraćamo ga i na njegovo mesto upisujemo AVAILABLE
 - inače vratimo None

- **put**(k, o)
 - izuzetak ako je tabela puna
 - počinjemo od ćelije $h(k)$
 - ispitujemo naredne ćelije sve dok se ne dogodi nešto od:
 - našli smo ćeliju koja je prazna ili sadrži AVAILABLE
 - isprobali smo svih N ćelija
 - upišemo (k, o) u ćeliju i

Heš mapa sa linearnim traženjem₁

```
class LinearHashMap(HashMap):
    """
    Klasa modeluje heš mapu koja rešava kolizije linearnim pretraživanjem
    """
    # sentinel koji označava lokaciju u mapi sa koje je obrisani element
    _MARKER = object()

    def _is_available(self, bucket_index):
        """
        Metoda proverava da li je bucket sa zadatim indeksom slobodan

        Argument:
        - `i`: indeks bucketa
        """
        return self._data[bucket_index] is None or self._data[bucket_index] is self._MARKER

    def _find_bucket(self, bucket_index, key):
        """
        Metoda pronalazi bucket koji sadrži element sa zadatim ključem
        """
        available_slot = None
        while True:
            if self._is_available(bucket_index):
                if available_slot is None:
                    available_slot = bucket_index

                if self._data[bucket_index] is None:
                    return False, available_slot

            elif key == self._data[bucket_index].key:
                return True, bucket_index

            bucket_index = (bucket_index+1) % len(self._data)
```

Heš mapa sa linearnim traženjem₂

```
def _bucket_getitem(self, bucket_index, key):
    """
    Pristup elementu u okviru bucketa

    Metoda najpre pristupa bucketu sa zadatim indeksom. Ukoliko bucket postoji, pretražuje se. Ako je element pronađen metoda vraća njegovu vrednostu, dok se u suprotnom podiže odgovarajući izuzetak.

    Argumenti:
    - `i`: indeks bucketa
    - `key`: ključ elementa
    """
    found, index = self._find_bucket(bucket_index, key)
    if not found:
        raise KeyError('Ne postoji element sa traženim ključem.')
    return self._data[index].value

def _bucket_setitem(self, bucket_index, key, value):
    """
    Postavljanje vrednosti elementa sa zadatim ključem

    Metoda najpre pronalazi bucket sa zadatim indeksom, a zatim dodaje novi element ili ažurira postojeći na osnovu zadatog ključa. Ukoliko bucket ne postoji, kreira se novi.

    Argumenti:
    - `i`: indeks bucketa
    - `key`: ključ elementa
    - `value`: vrednost elementa
    """
    found, available_bucket_index = self._find_bucket(bucket_index, key)
    if not found:
        self._data[available_bucket_index] = MapElement(key, value)
        self._size += 1
    else:
        self._data[available_bucket_index].value = value
```


Heš mapa sa linearnim traženjem₃

```
def _bucket_delitem(self, bucket_index, key):
    """
    Brisanje elementa sa zadatim ključem

    Metoda najpre pristupa bucketu sa zadatim indeksom. Ukoliko
    bucket postoji, pretražuje se. Ako je element pronađen vrši
    se njegovo brisanje, u suprotnom se podiže odgovarajući
    izuzetak.

    Argumenti:
    - `i`: indeks bucketa
    - `key`: ključ elementa
    """
    found, index = self._find_bucket(bucket_index, key)
    if not found:
        raise KeyError('Ne postoji element sa traženim ključem.')

    # označi element markerom
    self._data[index] = self._MARKER

def __iter__(self):
    total_buckets = len(self._data)
    for i in range(total_buckets):
        if not self._is_available(i):
            yield self._data[i].key

def items(self):
    total_buckets = len(self._data)
    for i in range(total_buckets):
        if not self._is_available(i):
            yield self._data[i].key, self._data[i].value
```

Duplo heširanje

- koristi se **sekundarna** heš funkcija $d(k)$
- prilikom kolizije element se smešta u prvu slobodnu ćeliju iz niza

$$(i + jd(k)) \bmod N \quad \text{za } j = 0, 1, \dots, N - 1$$

- sekundarna heš funkcija ne sme vratiti 0
- veličina tabele N mora biti prost broj da bi se mogle probati sve ćelije
- čest izbor za $d(k)$:
 $d(k) = q - (k \bmod q)$
 - $q < N$
 - q je prost broj
- rezultat $d(k)$ je u intervalu $[1, q]$

Duplo heširanje: primer

- heš tabela sa duplim heširanjem
 - $N = 13$
 - $h(k) = k \bmod 13$
 - $d(k) = 7 - (k \bmod 7)$
- dodajemo ključeve 18, 41, 22, 44, 59, 32, 31, 73

k	$h(k)$	$d(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	
32	6	3	6	
31	5	4	5	9 0
73	8	4	8	

0	1	2	3	4	5	6	7	8	9	10	11	12



31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Performanse heširanja

- u najgorem slučaju pretraga, dodavanje, uklanjanje traju $O(n)$
- najgori slučaj: kada su **svi** ključevi u koliziji
- faktor popune $\alpha = n/N$ utiče na performanse
- ako heševi liče na slučajne brojeve može se pokazati da je očekivani broj probanja prilikom dodavanja $1/(1 - \alpha)$

Performanse heširanja

- očekivano vreme izvršavanja svih operacija je $O(1)$
- u praksi heširanje je vrlo brzo ako faktor popune nije blizu 100%