

Mape, heš tabele, skip liste, skupovi

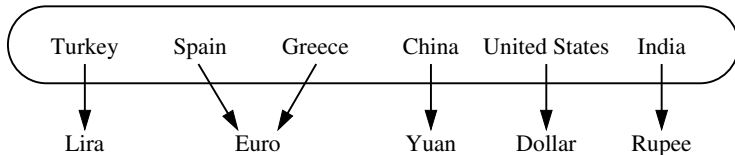
© Goodrich, Tamassia, Goldwasser

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

2021.

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 `KeyError`

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 $M2$

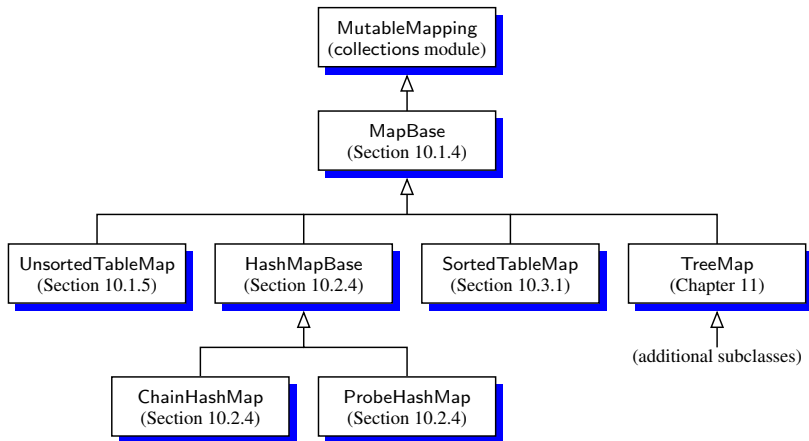
M == M2 vraća `True` ako mape sadrže iste parove (k,v)

M != M2 vraća `True` 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 MapBase

```

from collections import MutableMapping

class MapBase(MutableMapping):
    """Our own abstract base class that includes a nonpublic _Item class."""

    #-- nested _Item class --
    class _Item:
        """Lightweight composite to store key-value pairs as map items."""
        __slots__ = '_key', '_value'

        def __init__(self, k, v):
            self._key = k
            self._value = v

        def __eq__(self, other):
            return self._key == other._key    # compare items based on their keys

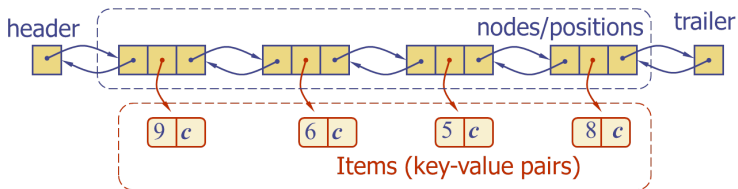
        def __ne__(self, other):
            return not (self == other)        # opposite of __eq__

        def __lt__(self, other):
            return self._key < other._key     # compare items based on their keys

```


Mapa pomoću liste

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



Mapa pomoću liste: implementacija ₁

```

class UnsortedTableMap(MapBase):
    """Map implementation using an unordered list."""

    def __init__(self):
        """Create an empty map."""
        self._table = []                # list of _Item's

    def __getitem__(self, k):
        """Return value associated with key k (raise KeyError if not found)."""
        for item in self._table:
            if k == item._key:
                return item._value
        raise KeyError('Key Error: ' + repr(k))

    def __setitem__(self, k, v):
        """Assign value v to key k, overwriting existing value if present."""
        for item in self._table:
            if k == item._key:           # Found a match:
                item._value = v          # reassign value
                return                   # and quit
        # did not find match for key
        self._table.append(self._Item(k,v))

```

Mapa pomoću liste: implementacija 2

```
def __delitem__(self, k):
    """Remove item associated with key k (raise KeyError if not found)."""
    for j in range(len(self._table)):
        if k == self._table[j]._key:
            self._table.pop(j)
            return
    raise KeyError('Key Error: ' + repr(k))

def __len__(self):
    """Return number of items in the map."""
    return len(self._table)

def __iter__(self):
    """Generate iteration of the map's keys."""
    for item in self._table:
        yield item._key
```

*# Found a match:
remove item
and quit*

yield the KEY

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 $(1, D), (3, Z), (6, C), (7, Q)$

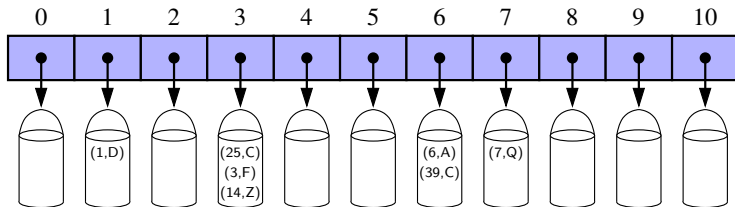
0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Koja su ograničenja ovog pristupa?

- operacije su $O(1)$

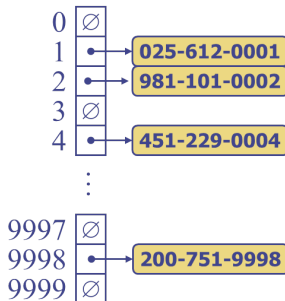
Hash tabela

- 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

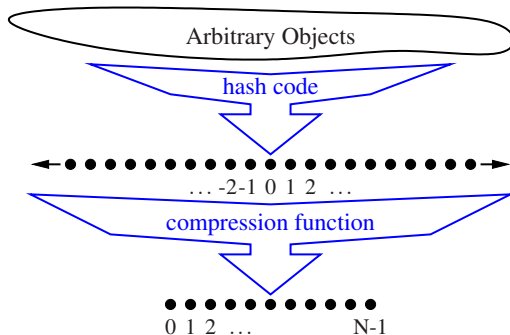


Hash funkcija

- **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 tabela ₁

```

class HashMapBase(MapBase):
    """Abstract base class for map using hash-table with MAD compression.

    Keys must be hashable and non-None.
    """

    def __init__(self, cap=11, p=109345121):
        """Create an empty hash-table map.

        cap        initial table size (default 11)
        p          positive prime used for MAD (default 109345121)
        """
        self._table = cap * [ None ]
        self._n = 0                                # number of entries in the map
        self._prime = p                            # prime for MAD compression
        self._scale = 1 + randrange(p-1)           # scale from 1 to p-1 for MAD
        self._shift = randrange(p)                # shift from 0 to p-1 for MAD

    def _hash_function(self, k):
        return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)

    def __len__(self):
        return self._n

```

Apstraktna heš tabela 2

```

def __getitem__(self, k):
    j = self._hash_function(k)
    return self._bucket_getitem(j, k)           # may raise KeyError

def __setitem__(self, k, v):
    j = self._hash_function(k)
    self._bucket_setitem(j, k, v)              # subroutine maintains self._n
    if self._n > len(self._table) // 2:        # keep load factor <= 0.5
        self._resize(2 * len(self._table) - 1) # number 2^x - 1 is often prime

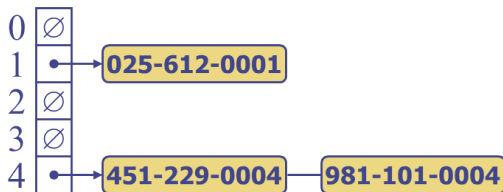
def __delitem__(self, k):
    j = self._hash_function(k)
    self._bucket_delitem(j, k)                 # may raise KeyError
    self._n -= 1

def _resize(self, c):
    """Resize bucket array to capacity c and rehash all items."""
    old = list(self.items())                    # use iteration to record existing items
    self._table = c * [None]                   # then reset table to desired capacity
    self._n = 0                                # n recomputed during subsequent adds
    for (k,v) in old:
        self[k] = v                           # reinsert old key-value pair

```


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



Mapa sa ulančavanjem

- delegiramo operacije mapi implementiranoj pomoću liste za svaku ćeliju

```
def get(k):  
    return A[h(k)].get(k)
```

```
def put(k, v):  
    t = A[h(k)].put(k, v)  
    if t is None:  
        n = n + 1  
    return t
```

```
def remove(k):  
    t = A[h(k)].remove(k)  
    if t is not None:  
        n = n - 1  
    return t
```

Heš tabela sa ulančavanjem ₁

```

class ChainHashMap(HashMapBase):
    """Hash map implemented with separate chaining for collision resolution."""

    def _bucket_getitem(self, j, k):
        bucket = self._table[j]
        if bucket is None:
            raise KeyError('Key Error: ' + repr(k))           # no match found
        return bucket[k]                                       # may raise KeyError

    def _bucket_setitem(self, j, k, v):
        if self._table[j] is None:
            self._table[j] = UnsortedTableMap()               # bucket is new to the table
        oldsize = len(self._table[j])
        self._table[j][k] = v
        if len(self._table[j]) > oldsize:                       # key was new to the table
            self._n += 1                                         # increase overall map size

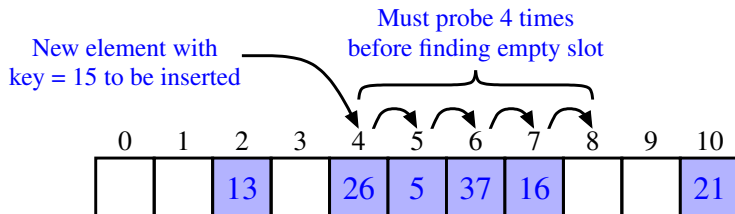
```

Heš tabela sa ulančavanjem ₂

```
def _bucket_delitem(self, j, k):  
    bucket = self._table[j]  
    if bucket is None:  
        raise KeyError('Key Error: ' + repr(k))    # no match found  
    del bucket[k]    # may raise KeyError  
  
def __iter__(self):  
    for bucket in self._table:  
        if bucket is not None:    # a nonempty slot  
            for key in bucket:  
                yield key
```

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 tabela sa linearnim traženjem ₁

```

class ProbeHashMap(HashMapBase):
    """Hash map implemented with linear probing for collision resolution."""
    _AVAIL = object()      # sentinel marks locations of previous deletions

    def _is_available(self, j):
        """Return True if index j is available in table."""
        return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL

    def _find_slot(self, j, k):
        """Search for key k in bucket at index j.

        Return (success, index) tuple, described as follows:
        If match was found, success is True and index denotes its location.
        If no match found, success is False and index denotes first available slot.
        """
        firstAvail = None
        while True:
            if self._is_available(j):
                if firstAvail is None:
                    firstAvail = j                # mark this as first avail
                if self._table[j] is None:
                    return (False, firstAvail)    # search has failed
            elif k == self._table[j]._key:
                return (True, j)                  # found a match
            j = (j + 1) % len(self._table)        # keep looking (cyclically)

```


Heš tabela sa linearnim traženjem ₂

```

def _bucket_getitem(self, j, k):
    found, s = self._find_slot(j, k)
    if not found:
        raise KeyError('Key Error: ' + repr(k))      # no match found
    return self._table[s]._value

def _bucket_setitem(self, j, k, v):
    found, s = self._find_slot(j, k)
    if not found:
        self._table[s] = self._Item(k,v)              # insert new item
        self._n += 1                                   # size has increased
    else:
        self._table[s]._value = v                     # overwrite existing

def _bucket_delitem(self, j, k):
    found, s = self._find_slot(j, k)
    if not found:
        raise KeyError('Key Error: ' + repr(k))      # no match found
    self._table[s] = ProbeHashMap._AVAIL              # mark as vacated

def __iter__(self):
    for j in range(len(self._table)):                  # scan entire table
        if not self._is_available(j):
            yield self._table[j]._key

```

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

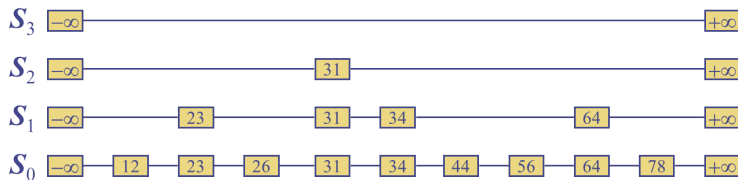
- očekivano vreme izvršavanja svih operacija je $O(1)$
- u praksi heširanje je vrlo brzo ako faktor popune nije blizu 100%

Skip lista

- binarna pretraga sa sortiranim nizom: omogućava pronalaženje elemenata u $O(\log n)$ vremenu
- ali su operacije dodavanja i uklanjanja $O(n)$ u najgorem slučaju
- **skip lista** omogućava sve u $O(\log n)$ vremenu
- u **prosečnom** slučaju

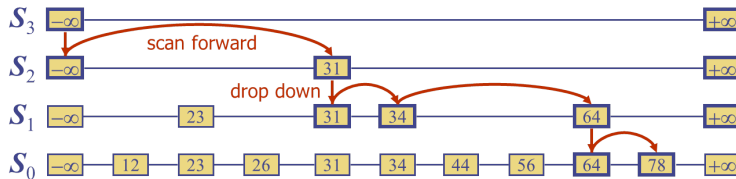
Skip lista

- **skip lista** za skup S elemenata (k, v) je serija lista S_0, S_1, \dots, S_h takvih da
 - 1 svaka lista S_i sadrži posebne ključeve $-\infty$ i ∞
 - 2 lista S_0 sadrži ključeve iz S u neopadajućem redosledu
 - 3 svaka lista je podskup prethodne
 $S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$
 - 4 lista S_h sadrži samo posebne ključeve



Skip lista: pretraga

- tražimo ključ x u skip listi na sledeći način:
- počnemo od prvog elementa liste na vrhu
- na tekućoj poziciji p_i poredimo x sa $y = key(next(p))$
 - $x = y$: pronašli smo traženi element
 - $x > y$: idemo napred (**scan forward**)
 - $x < y$: idemo dole (**drop down**)
- ako smo na nivou S_0 i treba da idemo dole, nema traženog elementa
- primer: tražimo 78



Algoritmi sa uvedenom slučajnošću

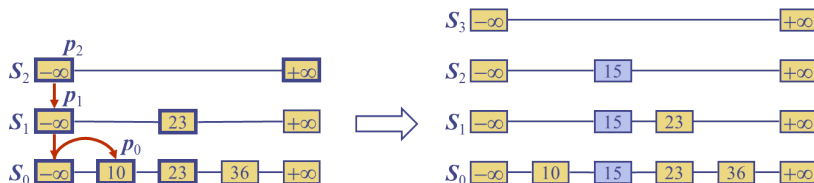
- randomized algorithms
- koriste (pseudo)slučajne vrednosti da upravlja svojim izvršavanjem
- sadrže naredbe nalik ovim:
$$b \leftarrow \text{random}()$$
if $b = 0$ **then**
 do A ...
else
 do B ...
- vreme izvršavanja zavisi od ishoda „bacanja novčića“

Algoritmi sa uvedenom slučajnošću

- analiza vremena izvršavanja ovakvih algoritama podrazumeva
 - svi ishodi „bacanja novčića“ su jednako verovatni
 - bacanja su međusobno nezavisna
- vreme izvršavanja **u najgorem slučaju** za ovakve algoritme je često veliko ali je vrlo malo verovatno (npr. sva bacanja novčića imaju isti ishod)
- koristićemo ovakav algoritam za dodavanje elemenata u skip listu

Dodavanje u skip listu

- dodavanje novog elementa (x, o) u skip listu:
 - ponavljamo bacanje novčića sve dok ne dobijemo „pismo“
 - sa i označimo broj puta koliko se pojavila „glava“
 - ako je $i \geq h$ dodaćemo nove liste S_{h+1}, \dots, S_{i+1} , svaku samo sa $-\infty + \infty$
 - tražimo x u skip listi i nađemo pozicije p_0, p_1, \dots, p_i elemenata sa najvećim ključem manjim od x u svakoj od lista S_0, S_1, \dots, S_i
 - za $j \leftarrow 0, \dots, i$ dodaćemo (x, o) u listu S_j nakon pozicije p_j
- primer: dodajemo ključ 15, za $i = 2$



Dodavanje u skip listu

SkipInsert(k, v)

Input: ključ k i vrednost v

Output: najviša pozicija dodatog elementa

$p \leftarrow \text{SkipSearch}(k)$

$q \leftarrow \text{None}$

{ q će biti najviši čvor u koloni novog elementa}

$i \leftarrow -1$

repeat

$i \leftarrow i + 1$

if $i \geq h$ **then**

$h \leftarrow h + 1$

{dodajemo novi nivo u skip listu}

$t \leftarrow \text{next}(s)$

$s \leftarrow \text{insertAfterAbove}(\text{None}, s, (-\infty, \text{None}))$

{povećaj levu}

$\text{insertAfterAbove}(s, t, (+\infty, \text{None}))$

{povećaj desnu kulu}

while $\text{above}(p)$ is **None** **do**

$p \leftarrow \text{prev}(p)$

{idi nazad}

$p \leftarrow \text{above}(p)$

{popni se 1 nivo}

$q \leftarrow \text{insertAfterAbove}(p, q, (k, v))$

{povećaj visinu kule}

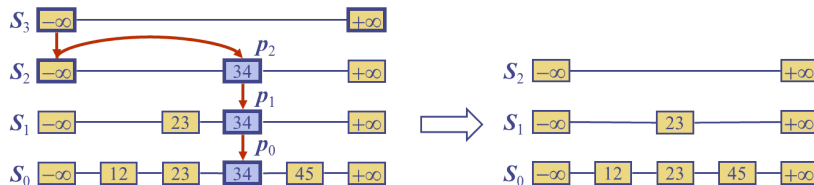
until $\text{coinFlip()} == \text{tails}$

$n \leftarrow n + 1$

return q

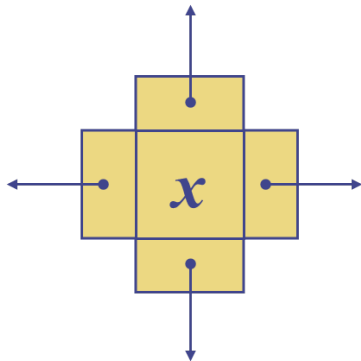
Uklanjanje iz skip liste

- uklanjanje elementa sa ključem x iz skip liste:
 - tražimo x u skip listi i nađemo pozicije p_0, p_1, \dots, p_i elemenata sa najvećim ključem manjim od x u svakoj od lista S_0, S_1, \dots, S_i
 - uklonimo pozicije p_0, p_1, \dots, p_i iz listi
 - uklonimo sve prazne liste osim jedne
- primer: uklanjamo ključ 34



Implementacija skip liste

- možemo da koristimo **quad-nodes**
 - element
 - link na prethodni
 - link na sledeći
 - link na čvor ispod
 - link na čvor iznad
- definišemo specijalne ključeve $PLUS_INF$ i $MINUS_INF$ i odgovarajući komparator



Skip liste i zauzeće prostora

- količina zauzete memorije zavisi od bacanja novčića
- iz teorije verovatnoće
 - (a) verovatnoća da se dobije i uzastopnih glava je $1/2^i$
 - (b) ako je svaki od n elemenata prisutan u listi sa verovatnoćom p , veličina skupa je np
 - (c) ako svaki od n događaja ima verovatnoću p , verovatnoća da će se desiti bar jedan nije veća od np
 - (d) očekivani broj bacanja novčića da se dobije „pismo“ je 2

Skip liste i zauzeće prostora

- posmatramo skip listu sa n elemenata
 - prema (a), dodaćemo čvor u listu S_i sa verovatnoćom $1/2^i$
 - prema (b), očekivana veličina liste S_i je $n/2^i$
- očekivani broj čvorova u skip listi je

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^i} < 2n$$

- prema tome, očekivani broj čvorova u skip listi sa n elemenata je $O(n)$

Visina skip liste

- vreme izvršavanja pretrage i dodavanja u skip listu zavisi od njene **visine**
- posmatramo skip listu sa n elemenata
 - prema (a), dodaćemo čvor u listu S_i sa verovatnoćom $1/2^i$
 - prema (c), verovatnoća da S_i ima bar jedan čvor je najviše $n/2^i$
- ako izaberemo $i = 3 \log n$, verovatnoća da $S_{3 \log n}$ ima bar jedan čvor je najviše

$$\frac{n}{2^{3 \log n}} = \frac{n}{n^3} = \frac{1}{n^2}$$

- prema tome, skip lista sa n elemenata je visoka najviše $3 \log n$ sa verovatnoćom najmanje $1 - 1/n^2$

Visina skip liste

- sa vrlo velikom verovatnoćom
- visina skip liste sa n elemenata je $O(\log n)$

Pretraga i ažuriranje skip liste

- vreme pretrage je proporcionalno
 - broju **drop down** koraka, plus
 - broju **scan forward** koraka
- drop down koraci su ograničeni visinom skip liste, dakle $O(\log n)$ sa velikom verovatnoćom

Pretraga i ažuriranje skip liste

- kada radimo **scan forward** korak, ključ se ne nalazi u listi iznad
 - scan forward postoji zato što je ranije novčić dao „pismo“
- prema (d), u svakoj listi očekivani broj scan forward koraka je 2
- prema tome, ukupan broj scan forward koraka je $O(\log n)$
- očekivano vreme za pretragu u skip listi je $O(\log n)$
- slično tome dodavanje i uklanjanje

Performanse skip liste

- skip lista je struktura podataka za implementaciju mape
- upotreba memorije je $O(n)$
- brzina pretrage, dodavanja i uklanjanja je $O(\log n)$ sa velikom verovatnoćom

Skup, multiskup, multimapa



- **skup** (set) je kolekcija elemenata koja ne poznaje redosled i ne sadrži duplikate
 - elementi skupa su nalik ključevima koji nemaju sebi asocirane vrednosti
- **multiskup** (multiset, **bag**) je skup koji dopušta duplikate
- **multimapa** je mapa koja dopušta da za jedan ključ bude vezano više vrednosti
 - indeks u knjizi preslikava pojam na jednu ili više stranica na kojima se on pominje

Skup ATP: osnovne operacije

<code>S.add(e)</code>	dodaje element e u S ; nema efekta ako je e već prisutan u S
<code>S.discard(e)</code>	uklanja e iz S ; nema efekta ako e nije prisutan u S
<code>e in S</code>	vraća <code>True</code> ako je e prisutan u S ; implementira je <code>__contains__</code>
<code>len(S)</code>	vraća broj elemenata u S ; implementira je <code>__len__</code>
<code>iter(S)</code>	iterira kroz elemente iz S ; implementira je <code>__iter__</code>

- pomoću osnovnih operacija implementiraju se sve ostale

Skup ATP: dodatne operacije

<code>S.remove(e)</code>	uklanja e iz S ; ako e nije prisutan u S izaziva <code>KeyError</code>
<code>S.pop()</code>	vraća i uklanja proizvoljan element iz S ; ako je S prazan izaziva <code>KeyError</code>
<code>S.clear()</code>	uklanja sve elemente iz S

Skup ATP: poređenje skupova

$S == T$	vraća True ako skupovi imaju jednak sadržaj
$S != T$	vraća True ako skupovi nemaju jednak sadržaj
$S \leq T$	vraća True ako je S podskup od T
$S < T$	vraća True ako je S pravi podskup od T
$S \geq T$	vraća True ako je S nadskup od T
$S > T$	vraća True ako je S pravi nadskup od T
$S.isdisjoint(T)$	vraća True ako su S i T disjunktni

Skup ATP: operacije nad skupovima

$S \mid T$	vraća novi skup koji je unija S i T
$S \mid= T$	ažurira S da bude unija S i T
$S \& T$	vraća novi skup koji je presek S i T
$S \&= T$	ažurira S da bude presek S i T
$S \wedge T$	vraća simetričnu razliku S i T
$S \wedge= T$	ažurira S da bude simetrična razlika S i T
$S - T$	vraća razliku S i T
$S -= T$	ažurira S da bude razlika S i T

Implementacija skupa: poređenje $S < T$

```
def __lt__(self, other):  
    """Vraća True ako je self pravi podskup od other."""  
    if len(self) >= len(other):  
        return False # pravi podskup mora biti manji  
    for e in self:  
        if e not in other:  
            return False # nije podskup, fali mu e  
    return True
```

Implementacija skupa: unija $S|T$

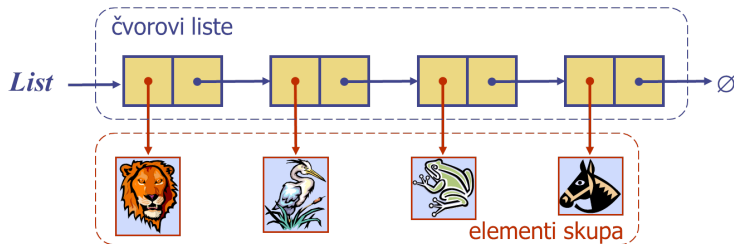
```
def __or__(self, other):  
    """Vraća novi skup kao uniju self i other."""  
    result = Set() # rezultat je nova instanca  
    for e in self:  
        result.add(e)  
    for e in other:  
        result.add(e)  
    return result
```

Implementacija skupa: operacija $S \mid = T$

```
def __ior__(self, other):  
    """Menja self da bude unija self i other."""  
    for e in other:  
        self.add(e)  
    return self    # mora da se vrati self za operator |=
```

Implementacija skupa: struktura podataka

- pomoću liste
- upotreba memorije je $O(n)$
- `__contains__` je $O(n)$



Implementacija skupa: struktura podataka

?

- pomoću hash tabele
- hash tabela čuva samo ključeve tj. elemente
- `__contains__` je $O(1)$
- za koje vreme rade presek i unija?