Bit Algo
START

Słowniki

Nazewnictwo

- słownik (dictionary)
- mapa (od ang. *map*, odwzorowanie)
- haszmapa (hashmapa)
- tablica z haszowaniem (hash table)
- tablica asocjacyjna/skojarzeniowa
- tablica mieszająca (trochę niepoprawne)

- można spotkać **niepoprawną** nazwę “tablica haszująca” - jest niepoprawna, bo to nie tablica haszuje, tylko funkcja haszująca

Słowniki ≈ uogólnione tablice

- tablica to ciągły obszar w pamięci, do którego możemy odwoływać się po numerycznym indeksie - dzięki temu komputer wie, o ile bajtów ma przejść w pamięci, np. arr[3]
- często chcielibyśmy indeksować nie numerem, a np. stringiem, przykładowo kiedy chcemy zliczać litery - odwołanie arr[4] += 1 dla inkrementacji licznika litery “d” jest mało czytelne, arr[“d”] += 1 dużo bardziej
- **słownik** działa w praktyce jak **tablica z uogólnionym indeksowaniem**

Jaką cenę za to płacimy?

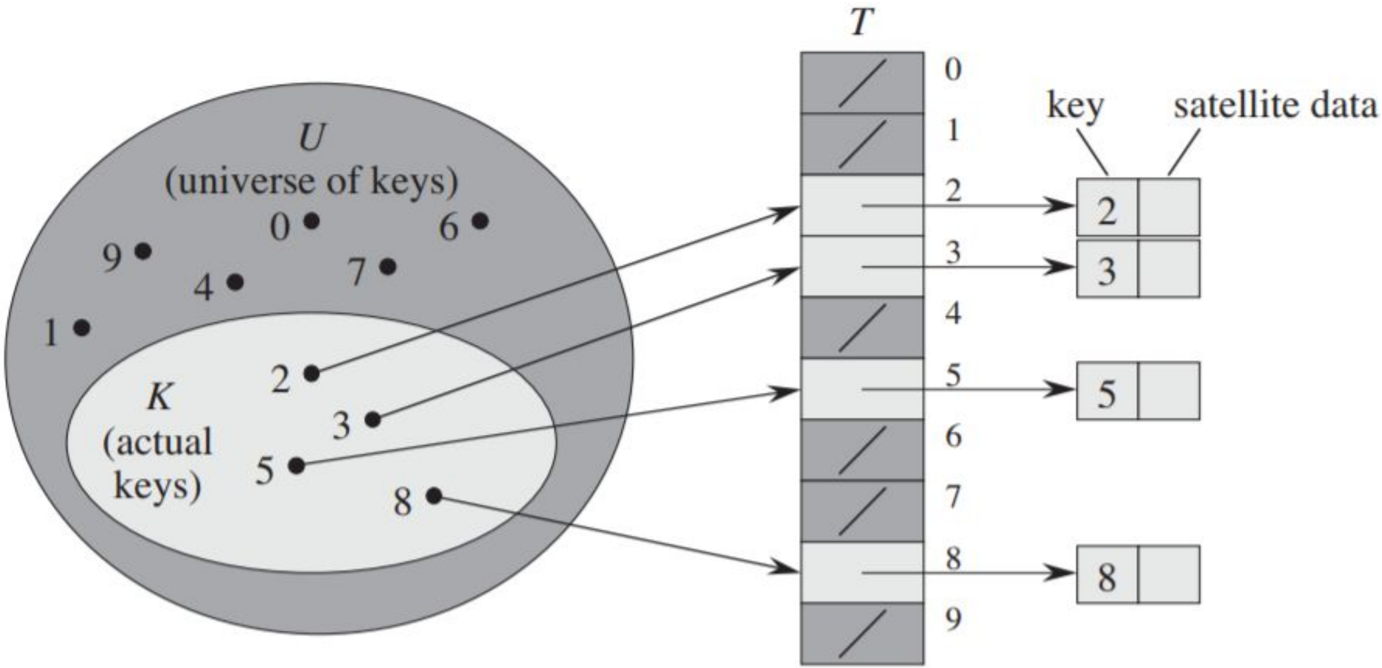
Czas dostępu	Tablica	Słownik
Średni	$\Theta(1)$	$\Theta(1)$
Pesymistyczny	$O(1)$	$O(n)$

- przeciętnie (**prawie zawsze**) słownik jest równie dobry, co tablica
- **ekstremalnie rzadko** słownik może mieć bardzo duży czas dostępu, pesymistycznie aż $O(n)$
- prawdopodobieństwo wyższych czasów dostępu bardzo szybko spada

Jak to działa pod spodem?

- słownik składa się z 2 elementów: **tablicy adresów (direct-access table)** oraz **funkcji haszującej**
- funkcja haszująca to translator: tłumaczy **klucz** (to, po czym się chcemy odwoływać do elementów) na **indeks** (pozycję w tablicy adresów)
- tablica ma na początek pola puste (np. None), przy wstawianiu wypełniamy je elementami, które zwykle mają postać (klucz, wartość)

Przykład


T
U
(universe of keys)
K
(actual keys)
0
1
2
3
4
5
6
7
8
9
key
satellite data
2
3
5
8

Funkcja haszująca

- najlepiej, żeby dawała jak najbardziej unikatowe indeksy dla różnych wartości - inaczej mamy **konflikt haszowania**
- powyższe jest równoznaczne z **generowaniem rozkładu jednostajnego**
- dobre funkcje: bitowy XOR i AND (^, &), przesunięcia bitowe (<< lub >>),

```
def hash(string):
    x = string[0] << 7
    for chr in string[1:]:
        x = ((1000003 * x) ^ chr)
        x = x & (1 << 32)
    return x
```

Konflikty haszowania

- problem: co, kiedy 2 klucze miałyby dostać to samo miejsce w tablicy haszowania?
- rozwiązania: mapowanie 1-do-1, listowe (chaining), probing, podwójne haszowanie

- najprostsze rozwiązanie: robimy funkcję mapującą 1-do-1, tzn. dla n w ogóle możliwych kluczy robimy tablicę rozmiaru n i konfliktów nigdy nie ma
- zalety: czas dostępu O(1)
- wady: wymagania pamięciowe O(n), rzadko znamy zakres kluczy

Chaining (rozwiązanie listowe)

- hash table zawiera **wskaźniki na listy**, a nie po prostu elementy
- listy zawierają pary (klucz, wartość), więc nawet, jeżeli 2 klucze dają ten sam hasz (indeks), to w tej liście możemy je rozróżnić, bo możemy po prostu porównywać klucze
- odwołanie do elementu wymaga dodatkowo przejścia po liście
- mniej konfliktów = krótsze listy = wydajniej
- zalety: prostota, dla dobrych funkcji haszujących wydajne
- wady: pesymistycznie czas dostępu to O(n), narzut pamięciowy na wskaźniki listy

Probing (rozwiązanie iteracyjne)

- po obliczeniu funkcji haszującej jeżeli pole jest już zajęte, to **iterujemy**, aż spotkamy to, co nas interesuje (puste pole dla wstawiania, wartość klucza dla wyszukiwania/usuwania)
- problem: wypełniamy z czasem hash table, zwiększając **współczynnik wypełnienia α (load factor)**, co wydłuża czas przeszukiwania
- rozwiązanie: definiuje się pewien próg α (np. 80%), po osiągnięciu którego przepisuje się cały hash table do większej tablicy (wymaga przehaszowania wszystkich elementów, więc przeciętnie Θ(liczba elementów))
- występuje problem **clusteringu**, czyli występowanie zajętych “obszarów” w tablicy

Przeszukiwanie liniowe (linear probing)

- idea: jeżeli obliczony indeks (i-ty indeks) jest zajęty, to “skaczemy” o ustalone z góry j, przeszukując po kolei i, i+j, i+2j etc. (jeżeli spotkamy koniec tablicy, to robimy %n)
- `index = (hash(key) + m * j) % n` , m - zmienna podczas iteracji, n - wielkość tablicy
- zaleta: prostota, wady: poniższe problemy
- **problem primary clustering,** z czasem budują się długie ciągi zapełnionych pól w pobliżu pierwotnych indeksów zwróconych przez funkcję haszującą, szczególnie, gdy mamy dużo konfliktów
- **problem cykliczności:** dla j > 1, jeżeli wielkość tablicy n jest podzielna przez j, to możemy cyklicznie wrócić na to samo miejsce po przejściu całej tablicy i “nie zauważyć” wolnego miejsca

Przeszukiwanie kwadratowe (quadratic probing)

- idea: zamiast skakać o j, skaczmy o pewną wartość funkcji kwadratowej:
- `index = (hash(key) + m * j + m * k`^{2}`) % n`

  m - zmienna podczas iteracji, n - wielkość tablicy, j, k - z góry ustalone stałe
- występuje **secondary clustering**, który jest mniej poważny, niż primary clustering
- zalety: względna prostota, clustering jest mniej poważny niż w przeszukiwaniu liniowym
- wady: dalej występuje clustering, ciężej jest dobrać współczynniki j oraz k

Podwójne haszowanie (double hashing)

- idea: po obliczeniu indeksu haszujemy klucz inną funkcją haszującą i przemieszczamy się o obliczoną przez nią wartość
- `index = (hash_1(key) + m * hash_2(key)) % n`

  m - zmienna podczas iteracji, n - wielkość tablicy, a, b - z góry ustalone stałe
- zalety: naprawdę rzadkie kolizje (obie funkcje musiałyby mieć kolizję dla tego samego klucza), brak problemu clusteringu
- wada: wartość hash_2(key) musi być **względnie pierwsza** do wielkości tablicy n; najprościej robi się to, zakładając $n = 2^z$ oraz wykorzystując funkcję haszującą hash_2 zwracającą zawsze nieparzystą liczbę

Haszowanie losowe (random hashing)

- idea: następny indeks do przeszukania oblicza się za pomocą **prostej** funkcji działającej jak generator liczb pseudolosowych (np. jakaś funkcja rekurencyjna)
- ważna jest **pseudo**losowość, bo dzięki temu dla danej liczby funkcja zwraca zawsze tę samą wartość
- zaleta: brak clusteringu - szanse na to, że klucze będą zachowywać się w sposób pseudolosowy, jest naprawdę pomijalna
- wada: trudność implementacji
- **używane w praktyce** - np. w Pythonie

Haszowanie losowe (random hashing)

- idea: następny indeks do przeszukania oblicza się za pomocą **prostej** funkcji działającej jak generator liczb pseudolosowych (np. jakaś funkcja rekurencyjna)
- dość podobne do double hashingu
- ważna jest **pseudo**losowość, bo dzięki temu dla danej liczby funkcja zwraca zawsze tę samą wartość
- zaleta: brak clusteringu - szanse na to, że klucze będą zachowywać się w sposób pseudolosowy, jest naprawdę pomijalna
- wada: trudność implementacji
- **używane w praktyce** - np. w Pythonie

Para liczb sumujących się do danej liczby

Problem: Mając daną na wejściu tablicę liczb naturalnych napisać procedurę odpowiadającą na pytanie, czy istnieje para liczb sumujących się do zadanej innej liczby. Przykład: [12, 3, 4, 90, 15, 55], liczba = 19 -> tak: 15 + 4.

Para liczb sumujących się do zadanej liczby

Rozwiązanie: należy każdą liczbę umieścić w słowniku, a następnie dla każdej liczby sprawdzić, czy w słowniku jest różnica między zadaną liczbą a nią. Czas: O(n), pamięć O(n).

Para liczb sumujących się do zadanej liczby

```
def findPair(arr, x):
    dictionary = set(arr)
    for i in arr:
        if x - i in dictionary:
            return True
    return False
```

Najdłuższy podciąg kolejnych liczb naturalnych

Problem: Mając na wejściu tablicę liczb naturalnych znaleźć długość najdłuższego podciągu kolejnych liczb naturalnych. Algorytm powinien działać w czasie O(n). Przykład: [1,5,3,4,8,10,12,11], odpowiedzią jest 3. Ciągiem może być 3,4,5 lub 10,11,12

Najdłuższy podciąg kolejnych liczb naturalnych

```
def longestConsecutiveSubsequence(arr):
    D = set(arr)
    result = 0
    for i in arr:
        if i-1 not in D:
            j = i
            while j in D:
                j +=1
            ans = max(ans, j-i)
    return ans
```

Losowy element ze słownika

Zaprojektuj strukturę danych przechowującą liczby naturalne, udostępniającą następujący Interfejs:

insert(num) - umieszcza w strukturze liczbę naturalną.

delete(num) - usuwa ze struktury liczbę

find(num) - sprawdza czy liczba jest w strukturze.

getRandom() - zwraca losową liczbę ze struktury. Do tej funkcji możesz wykorzystać funkcję randInt(N), któr w czasie O(1) zwraca losową liczbę naturalną z zakresu od 0 do N-1.

Wszystkie powyższe funkcje powinny działać w czasie O(1) zawsze!!!.

Losowy element ze słownika

Rozwiązanie: Mając randInt(N) można łatwo losować liczbę z tablicy, ponieważ wystarczy wylosować indeks poprzez randInt(len(array)). Dlatego w rozwiązaniu łączymy słownik ze zwykłą tablicą. Kluczami w słowniku będą przechowywane liczby. Dodatkowo liczby te będziemy przechowywać w tablicy i indeks po jakim się znajdują w tablicy będzie wartością pod odpowiednim kluczem w słowniku.

Losowy element ze słownika

insert(num) - wstawiamy element na koniec tablicy, a potem do słownika. Pod kluczem num umieszczam wartość len(array)-1

find(num) - tak jak wyszukiwanie w słowniku

delete(num) - znajdujemy pod jakim indeksem w tablicy znajduje się num i umieszczamy tam wartoś ostatniego elementu tablicy, potem podmieniamy w słowniku indeks pod, którym znajduje się ostatni elemen w tablicy. Na końcu usuwamy ostatni element tablicy. (jest to O(1), tylko usuwanie ze środka daje O(n)).

getRandom() - losujamy indeks z tablicy.

Losowy element ze słownika

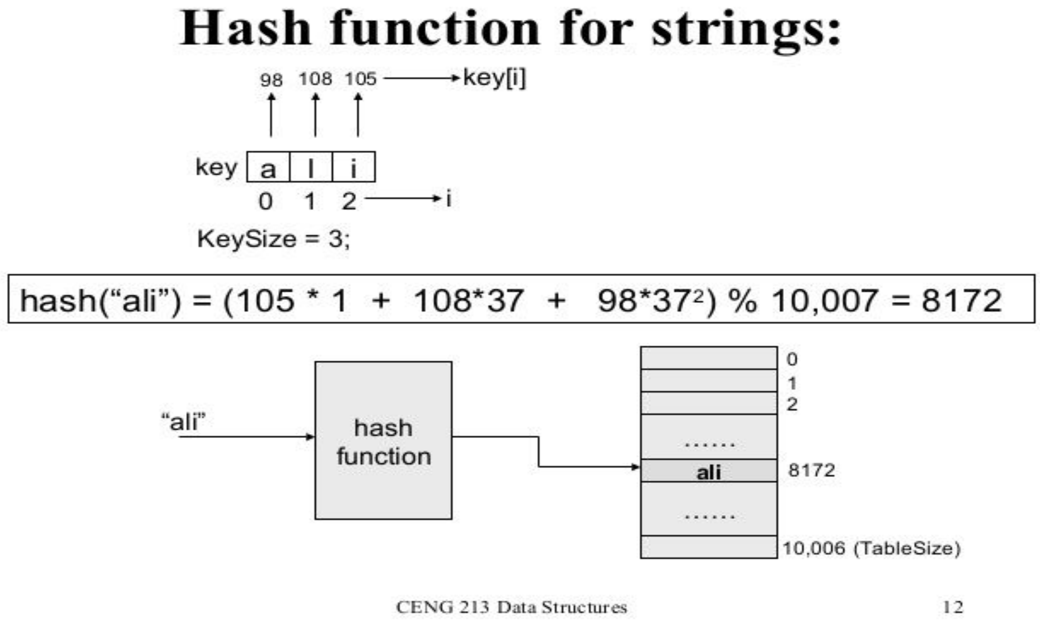
```
class RandomHashSet():
    def _init_(self):
        self.array = []
        self.d = {}
    def insert(self, num):
        self.array.append(num)
        self.d[num] = len(self.array)-1
    def find(self, num):
        return num in self.d
    def delete(self,  num):
        index = self.d[num]
        last = self.array[len(array)-1]
        self.array[index] = last
        del self.d[num]
        self.array.pop()
    def getRandom(self):
        return self.array[randInt(len(self.array))]
```

Zastosowanie haszowania - algorytm Rabina - Karpa

Problem: mając dany tekst T i wzorzec P odpowiedzieć na pytanie, czy T zawiera P. Przykład: T = aadghhhhhjukipooooopl, P = dghhh, odpowiedź: tak, a dla wzorca P = aadgc, odpowiedź: nie.

Zastosowanie haszowania - algorytm Rabina - Karpa

Propozycja: jeżeli funkcja hashująca zwróci różną wartość dla stringów, to na pewno są one różne. Jeśli taką samą - to jest to powód, żeby żeby je porównać. Zamiast sprawdzać każde podsłowo o długośc len(P), można sprawdzać te, które dają taki sam wynik funkcji hashującej. Możemy skorzystać z faktu, że jeżeli znamy hash dla T[i, j], to hash dla T[i+1, j+1] możemy wyznaczyć w czasie O(1).



Zastosowanie haszowania - dla zainteresowanych

Mając na wejściu tekst T zwróć największe takie k, że:

- istnieje rozkład na słowa: s1 + s2 + s3 + .. + sk = T, ai != “”
- dla każdego 1 <= i <= k, zachodzi ai = a(k+1-i)
- algorytm działa w oczekiwanym czasie O(n)

Bit Algo

START