



STRUKTURE PODATAKA I ALGORITMI

Predavanje 14


Ishod 6

1

Rječnici

- Rječnik (engl. *dictionary, associative array, map, symbol table*) je kontejner koji sadrži kolekciju parova (ključ, vrijednost) i koji pruža operacije:
 - Dodavanje novog para
 - Uklanjanja para
 - Modifikaciju vrijednosti postojećeg para (ali ne i ključa)
 - Dohvat vrijednosti prema zadanom ključu (naglasak)
- U nekim programskim jezicima (Python) su ugrađeni tipovi
- Dva glavna smjera implementacije rječnika su:
 - Hash tablice
 - Binarna stabla traženja (ishod 4)

Strana • 2



2

Usporedba smjerova implementacija rječnika

- Naglasak na brzom pretraživanju, umetanju i brisanju

Underlying data structure	Lookup		Insertion		Deletion		Ordered
	average	worst case	average	worst case	average	worst case	
Hash table	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	No
Self-balancing binary search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	Yes
unbalanced binary search tree	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	Yes
Sequential container of key-value pairs (e.g. association list)	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	No

Rječnici stablima	Rječnici <i>hash</i> tablicama
Elementi su sortirani	Elementi nisu sortirani
Troši manje memorije	Troši više memorije
Bolje ako ima umetanja i/ili brisanja	Bolje ako dominira pretraživanje
Garantirane performanse	Performanse mogu varirati

Strana • 3

Preuzeto s:
en.wikipedia.org

3

TABLICE S DIREKTNIM ADRESIRANJEM

Strana • 4



4

Uvod

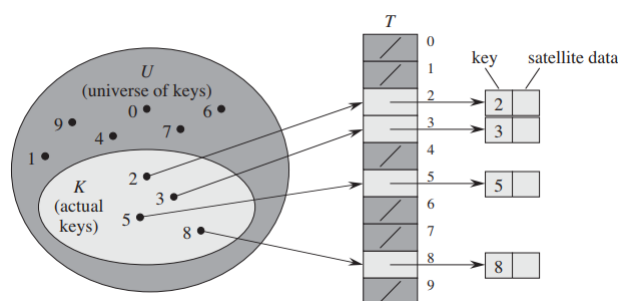
- **Tablice s direktnim adresiranjem** (engl. *direct-address tables*) su poseban, jednostavan slučaj *hash* tablica
 - Odlične ako imamo relativno mali broj jedinstvenih ključeva i ako razlika najmanjeg i najvećeg ključa nije prevelika
 - Ne koristi se u praksi
- Osnova tablica s direktnim adresiranjem je polje
 - Mjesto u polju na nekom indeksu se naziva **slot** ili **bucket**
 - Svaki mogući ključ ima za sebe rezervirano jedno mjesto u polju
 - Indeks rezerviranog mjesta je jednak ključu

Strana • 5



5

Primjer tablice s direktnim adresiranjem



- **Varijacije:**
 - Ponekad u *slotu* umjesto pokazivača čuvamo izravno podatke
 - Ponekad uopće ne čuvamo ključ jer je sâm indeks u stvari ključ
 - Potrebno pravilno definirati prazan *slot*

Strana • 6

Preuzeto iz Cormen et al:
Introduction to Algorithms

6

Operacije

- Kako bismo implementirali tri rječničke operacije:
 - SEARCH(key)
 - return polje[key]
 - INSERT(key, value)
 - polje[key] ← value
 - DELETE(key)
 - polje[key] ← NULL
- Kolika je složenost svake operacije?
 - $O(1)$
 - Fantastično, ali nepraktično za realne uvjete: potrebno veliko, vjerojatno slabo iskorišteno polje (npr. ključevi 2, 7 i 545.000)

Strana • 7



7

HASH TABLICE

Strana • 8



8

Hash tablice

Underlying data structure	Lookup		Insertion		Deletion		Ordered
	average	worst case	average	worst case	average	worst case	
Hash table	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	No

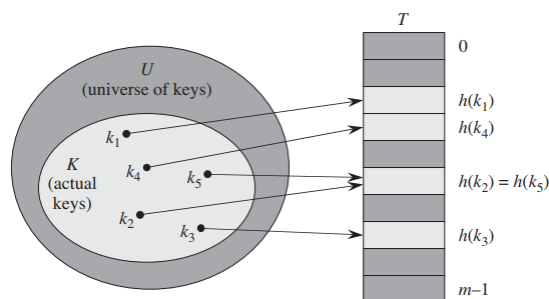
- Hash tablice imaju konstantnu složenost samo u najboljem i srednjem slučaju
 - U najgorem slučaju daju neprihvatljivu linearnu složenost
- Ideja: element s ključem key ćemo smjestiti na indeks $h(key)$
 - Koristimo **hash funkciju** h kako bismo od ključa izračunali indeks
 - To nam omogućava da imamo polje koje je znatno manje od ukupnog broja ključeva
- Hash funkcija može više ključeva pretvarati u isti indeks, što se naziva **kolizija** (engl. *collision*)

Strana • 9



9

Primjer hash tablice



- Pronađite koliziju

Strana • 10

Preuzeto iz Cormen et al:
Introduction to Algorithms

10

Hash funkcije

- Hash funkcija je funkcija koja pretvara (*hashira*) zadani ključ u indeks polja
 - S jedne strane spektra su tzv. tablice s direktnim adresiranjem
 - $h(\text{key}) = \text{key}$
 - Indeks je jednak ključu (svaki ključ ima "rezervirano" mjesto)
 - Potrebna su velika, potencijalno slabo iskorištena polja
 - Druga krajnost bi bila funkcija $h(\text{key}) = x$
 - Sve ključeve pretvara u isti indeks x (kolizije)
- Dobre *hash* funkcije su negdje između tih krajnosti
 - One ravnomjerno pretvaraju ključeve u indekse tako da u svaki indeks bude pretvoren otprilike jednak broj ključeva

Strana • 11



11

Rješavanje kolizija

- Pošto *hash* funkcija može više ključeva pretvoriti u isti indeks, potrebno je o tome voditi računa
- Glavni načini **rješavanja kolizija**:
 - Kod tablica s direktnim adresiranjem kolizija je izbjegnuta funkcijom $h(\text{key}) = \text{key}$
 - Ulančavanje (engl. *chaining*)
 - Otvoreno adresiranje (engl. *open addressing*)

Strana • 12



12

ULANČAVANJE

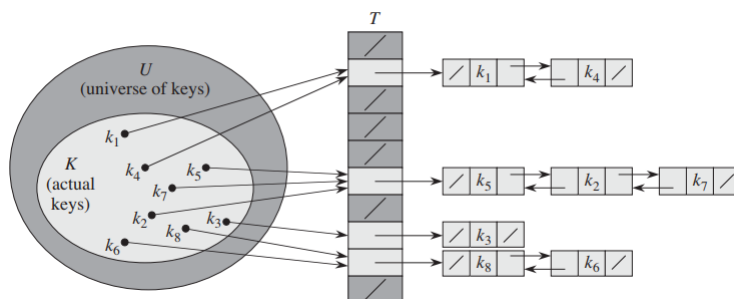
Strana • 13



13

Ulančavanje

- Kod ulančavanja, sve ključeve koji se *hashiraju* u isti *slot* stavljamo u **povezanu listu** koja pripada tom slotu
 - Svaki slot sadrži ili povezanu listu ili nullptr



Strana • 14

Preuzeto iz Cormen et al:
Introduction to Algorithms

14

Operacije

- Pogledajmo kako bismo na *hash* tablici s ulančavanjem implementirali tri rječničke operacije:
 - SEARCH(key)
 - Izračunaj *hash* vrijednost na osnovu ključa
 - Pronađi i vrati zapis sa zadanim ključem iz liste u izračunatom *slotu*
 - INSERT(key, value)
 - Izračunaj *hash* vrijednost na osnovu ključa
 - U listu u izračunatom *slotu* umetni ključ i vrijednost
 - DELETE(key)
 - Izračunaj *hash* vrijednost na osnovu ključa
 - Ukloni zapis sa zadanim ključem iz liste u izračunatom *slotu*

Strana • 15



15

Primjer

- Pretpostavimo da imamo *hash* tablicu s poljem od 7 elementa te *hash* funkcijom: $h(\text{key}) = \text{key} \bmod 7$. Nacrtajmo kako izgleda *hash* tablica nakon umetanja podataka:

1	Danijel Subašić	15	Ivan Perišić
2	Lovre Kalinić	16	Mateo Kovačić
3	Dominik Livaković	17	Marko Rog
4	Vedran Ćorluka	18	Marcelo Brozović
6	Domagoj Vida	19	Milan Badelj
7	Ivan Strinić	20	Mario Mandžukić
8	Šime Vrsaljko	21	Nikola Kalinić
9	Josip Pivarić	22	Andrej Kramarić
10	Tin Jedvaj	23	Marko Pjaca
11	Dejan Lovren		
12	Matej Mitrović		
13	Luka Modrić (C)		
14	Ivan Rakitić		

- Što bi se promijenilo da koristimo $h(\text{key}) = 0$?

Strana • 16



16

Performanse (1/2)

- Performanse umetanja:
 - Ako dopuštamo duple ključeve: $O(1)$
 - Ako ne dopuštamo duple ključeve: $O(k)$, gdje je k broj elemenata u slotu
 - Moramo pregledati sve elemente u slotu i provjeriti postoji li ključ
 - Ako znamo da je ključ jedinstven: $O(1)$
- Performanse brisanja:
 - Općenito: $O(k)$
 - Ako imamo iterator na element i i ako je lista dvostruka: $O(1)$

Strana • 17



17

Performanse (2/2)

- Performanse pretraživanja:
 - Definiramo **faktor opterećenja** (engl. *load factor*): $\alpha = \frac{n}{m}$
 - n je broj elemenata
 - m je broj slotova
 - Preduvjet 1: dok god je faktor opterećenja oko ili ispod 1, pretraživanje će u prosječnom slučaju biti $\Theta(1)$
 - $n \ll m \Rightarrow \alpha$ pada prema 0 \Rightarrow raste neiskorištenost prostora
 - $n \gg m \Rightarrow \alpha$ raste \Rightarrow performanse padaju
 - $n \approx m \Rightarrow \alpha$ oko 1 \Rightarrow optimum
 - Preduvjet 2: *hash* funkcija radi dobar posao
 - Inače nam faktor opterećenja ne igra nikakvu ulogu

Strana • 18



18

Zadatak

- Uzmimo ključeve od 1 do 100 i smjestimo u vlastitu jednostavnu implementaciju *hash* tablice s ulančavanjem (vrijednost neka bude kvadrat ključa). Koristimo *hash* funkciju $h(\text{key}) = \text{key} \bmod 31$. Ispišimo raspodjelu ključeva po *bucketima* te demonstrirajmo traženje.

Strana * 19



19

Rješenje – main

```
int main() {
    hash_table ht;
    for (int i = 1; i <= 100; i++) {
        ht.insert(i, i*i);
    }

    ht.print();

    int n;
    cout << "Upisite broj: ";
    cin >> n;
    cout << "Kvadrat broja je: " << ht.search(n) << endl;

    return 0;
}
```

Strana * 20



20

Rješenje – zaglavlje

```

struct entry {
    int key;
    int value;
    entry(int key, int value) {
        this->key = key;
        this -> value = value;
    }
};

class hash_table {
private:
    list<entry> ARRAY[31];
    int h(int key);
public:
    void insert(int key, int value);
    int search(int key);
    void print();
};

```

Strana * 21



21

Rješenje – implementacija (1/2)

```

int hash_table::h(int key) {
    return key % 31;
}

void hash_table::insert(int key, int value) {
    int slot = h(key);
    ARRAY[slot].push_back(entry(key, value));
}

void hash_table::print() {
    for (int i = 0; i < 31; i++) {
        cout << "Slot " << i << ": ";
        for (auto it = ARRAY[i].begin(); it !=
            ARRAY[i].end(); ++it) {
            cout << "key=" << it->key << " ";
        }
        cout << endl;
    }
}

```

Strana * 22



22

Rješenje – implementacija (2/2)

```
int hash_table::search(int key) {  
    int slot = h(key);  
    for (auto it = ARRAY[slot].begin(); it !=  
        ARRAY[slot].end(); ++it) {  
        if (it->key == key) {  
            return it->value;  
        }  
    }  
    return -1;  
}
```