RI301
Strukture podataka

dr.sc. Edin Pjanić

Pregled predavanja

- Problem pretraživanja strukture podataka
- Raspršeno adresiranje – hashing
    - hash funkcije
    - problem kolizije

Složenost pronalaska elementa

- Pronalazak elementa u linearnim strukturama podataka zahtijeva vrijeme proporcionalno broju ulaznih podataka (linearna složenost), odnosno O(n)
- Korištenjem BST postiže se O(log n), ali najgori slučaj je ipak O(n)
    - Da bismo garantovali složenost O(log n), moramo BST balansirati (nismo razmatrali)

- Efikasnost (složenost) nabrojanih pristupa ovise o broju podataka u kontejneru.
- Metodi pretraživanja čija složenost ne ovisi o broju podataka bi bili bolji.
    - Npr. pristup elementu vektora (niza) po indeksu je O(1)

Pretraživanje - primjer

- Razmotrimo sljedeću C++ klasu koja opisuje zapis podataka o studentima:

```
class Student {
   string ime;                    // ime studenta
   string opstina;                // opstina rodjenja
   long id;                       // jedinstveni identifikator
   std::vector<Ocjena> ocjene; // ocjene studenta
};
```

- Polje **`id`** se može koristiti kao ključ pri traženju studenta u odgovarajućim zapisima unutar kontejnera.
  - Identifikator je 5 cifreni broj (npr. 18553, 93322, 33452, itd.)
- Pretpostavimo da želimo smjestiti 10000 zapisa u nekom kontejneru. Koju vrstu kontejnera upotrijebiti za najbrži pristup?

Raspršeno adresiranje - hashing

- U prethodnom primjeru sa 10,000 studenata:

- Implementacija kontejnera linkanom listom bi za pristup (pretragu) po id-u zahtijevala O(n).
- Balansirano stablo BST bi bilo O(log n)
- Niz (vektor) od 10,000 elemenata sortiranih po polju id uz korištenje binarnog pretraživanja: O(log n)
- O(1) bismo mogli postići ako bismo koristili niz tako da nam polje **id** bude indeks. Imali bismo direktan pristup.
  - u tu svrhu bi nam trebao niz od 100,000 elemenata => ogromna količina neiskorištene memorije (90,000)

- Postoji li način za postizanje pibližno O(1) bez mnogo neiskorištene memorije?
  - Raspršeno adresiranje - HASHING

Hashing – nastavak

- Pretraživanje korištenjem tehnika hashinga uključuje:
    - računanje hash funkcije koja transformiše ključ u indeks (adresu) unutar tabele (niza).
    - rješavanje slučajeva kolizije
- Pri korištenju pristupa raspršenog adresiranja pokušava se pronaći balans između efikasnog korištenja razumne količine memorije i brzog pristupa memoriji (elementima) kontejnera.
- Pristup se bazira na tome da se podaci (zapisi) čuvaju u **tabeli** (nizu) u internoj ili eksternoj memoriji a za dobijanje **lokacije** (adrese) zapisa u toj tabeli koristi se **hash funkcija** koja se računa nad **ključem** traženog zapisa.
- Engl.: hash tables, scatter tables

Primjer 1 – ilustracija hashinga

- Neka imamo sljedeći skup zapisa o studentima:

Ime	Opština	ID
Vladislav	Tuzla	16113
Jasmin	Lukavac	14024
Medina	Živinice	14113
Želimir	Tuzla	17080
Emir	Kalesija	18010
Belma	Srebrenik	15134
Emina	Gračanica	15133

- Koristićemo hash funkciju h(s) = s.id % 13 da bismo ove podatke smjestili u niz veličine 13. (hash funkcijom dobijamo indeks)

Primjer 1 – nastavak

- Dobićemo sljedeće mapiranje:

Ime	ID studenta	h(s) = id % 13
Vladislav	16113	
Jasmin	14024	
Medina	14113	
Želimir	17080	
Emir	18010	
Belma	15134	
Emina	15133	

0	1	2	3	4	5	6	7	8	9	10	11	12
	Emina	Belma			Emir	Vladislav		Medina		Jasmin	Želimir	

Hash funkcije

- Hash funkcija **h** je funkcija koja transformiše ključ iz skupa **K** u adresu hash tabele od **n** elemenata:

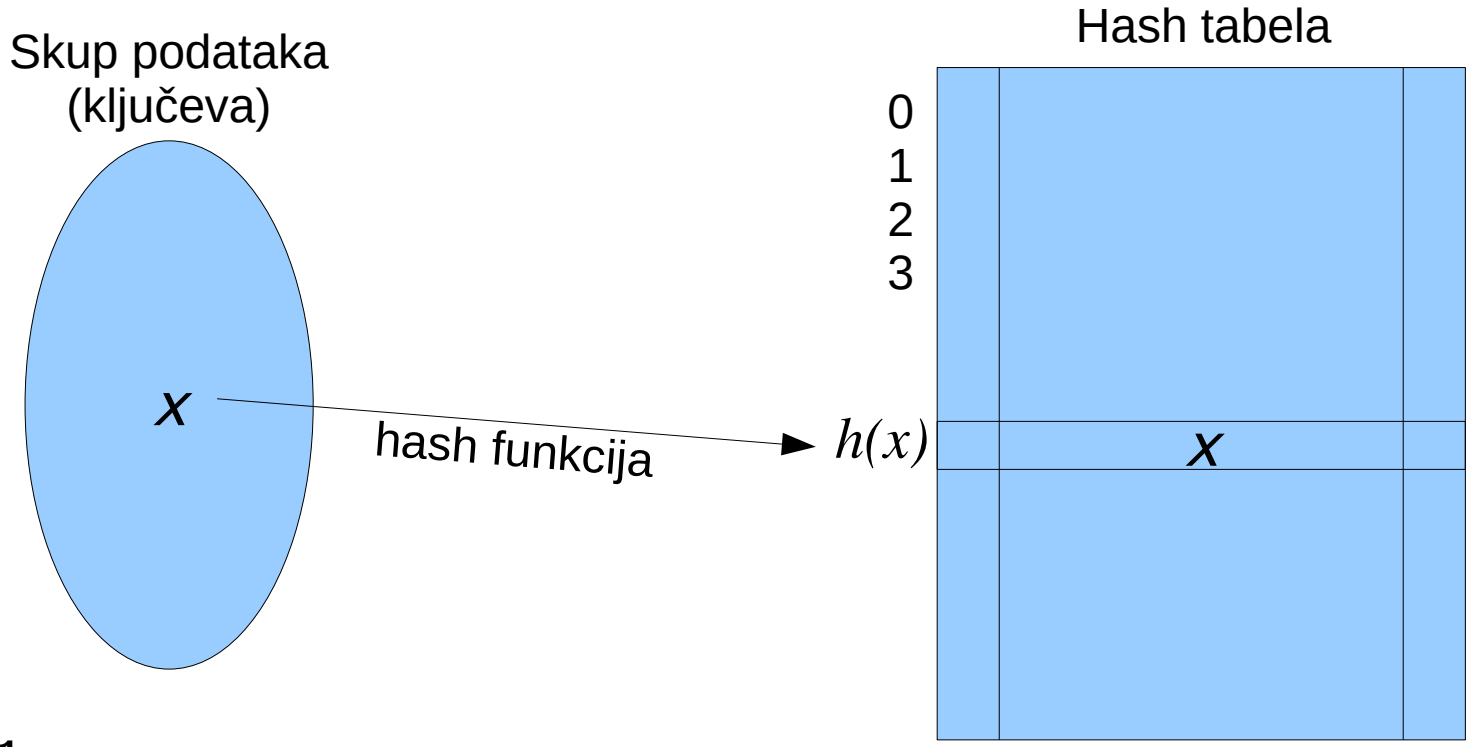
```
h: K -> {0, 1, ..., n-2, n-1}
```

- Ključ može biti broj, string ili bilo koji drugi tip podatka.
- Veličina skupa svih mogućih vrijednosti ključeva **K** je obično velika.
- Moguće je da funkcija **h** za različite ključeve vrati istu vrijednost.
  - Ovakva situacija se zove **kolizija** a odgovarajući ključevi se zovu **sinonimi**.

Jesu li kolizije korisne/poželjne?

Hash funkcije - nastavak

- Dobra hash funkcija bi trebala:
  - biti laka i brza za izračunavanje
  - minimizirati broj kolizija
  - raspršiti vrijednosti ključeva po hash tabeli ravnomjerno
  - koristiti sve informacije iz ključa



Česte hash funkcije

Ostatak cjelobrojnog dijeljenja, pri čemu se koristi veličina tabele kao djelilac:

- Računa hash vrijednost od ključa korištenjem operatora %
- Veličinu tabele (niza) koja je potencija broja 2 , kao npr. 32, 64, 1024 itd, bi trebalo izbjegavati jer vodi ka više kolizija.
- Također treba izbjegavati potencije broja 10 kod ključeva koji su bazirani na decimalnom brojnom sistemu.
- Dobar izbor veličine tabele su prosti brojevi koji nisu blizu potencija broja 2.

```
int h(int x, int D)
{
  return x % D;
}
```

Česte hash funkcije

Odbacivanje ili ekstrakcija cifre/karaktera:

- Bazirana je na raspodjeli cifara ili karaktera unutar ključa.
- Iz ključa se uzimaju cifre koje su bolje raspodijeljene i koriste se za računanje hash funkcije.
- Npr. ID studenta, broj telefona ili JMBG može sadržavati neke zajedničke dijelove koji povećavaju mogućnost kolizije.
- Ovakva hash funkcija je vrlo brza, ali često raspodjela cifara u ključu nije baš ravnomjerna.

Česte hash funkcije

Konverzija baze:

- Transformacija ključa u drugi brojni sistem kako bi se dobila hash vrijednost.
- Obično se za računanje hash adrese ne koriste baze 10 i 2.
- Npr. za mapiranje ključa 55354 u opseg 0 do 9999 koristeći bazu 11 imamo:
  - $55354_{10}=38652_{11}$.
- Možemo odbaciti cifru najveće težine (3) što dovodi do broja 8652 koji predstavlja hash adresu u opsegu 0 - 9999.

Česte hash funkcije

Sredina kvadrata:

- Ključ se kvadrira a kao hash vrijednost se uzima odgovarajući broj cifara iz sredine rezultata.
- Npr. za mapiranje ključa 3121 u hash tabelu veličine 1000, kvadriramo 3121 => 3121^2 = 9740641 i uzimamo 406 kao ključ.
- Radi dobro ako ključevi nemaju mnogo nula na svom početku ili kraju.
- Ključevi koji nisu cijeli brojevi se prethodno moraju obraditi da bi se dobila odgovarajuća cjelobrojna vrijednost.

Česte hash funkcije

Bilo koji pristup koji vodi ka što ravnomjernijem i pseudoslučajnom raspršivanju hash vrijednosti po cijelom opsegu.

- Npr. može se koristiti i shiftanje dijelova ključa da se dobije što "slučajnija" hash vrijednost. Na kraju se gotovo uvijek računa ostatak cjelobrojnog dijeljenja sa veličinom tabele:

```
long hash(string kljuc, int D) {
  long h=0;
  for (int i=0, i<kljuc.length(); i++)
  {
    h = (h << 4) | ( h >> 28);
    h += (long) kljuc[i];
  }
  return h % D;
}
```

Rješavanje kolizije

- Ako imamo preslikavanje N različitih vrijednosti (ključeva) u D različitih vrijednosti (ćelija tabele), pri čemu je N>D neizbježna je mogućnost kolizije.
- Npr. h(x) = x % 7
  - h(46)=4
  - h(60)=4

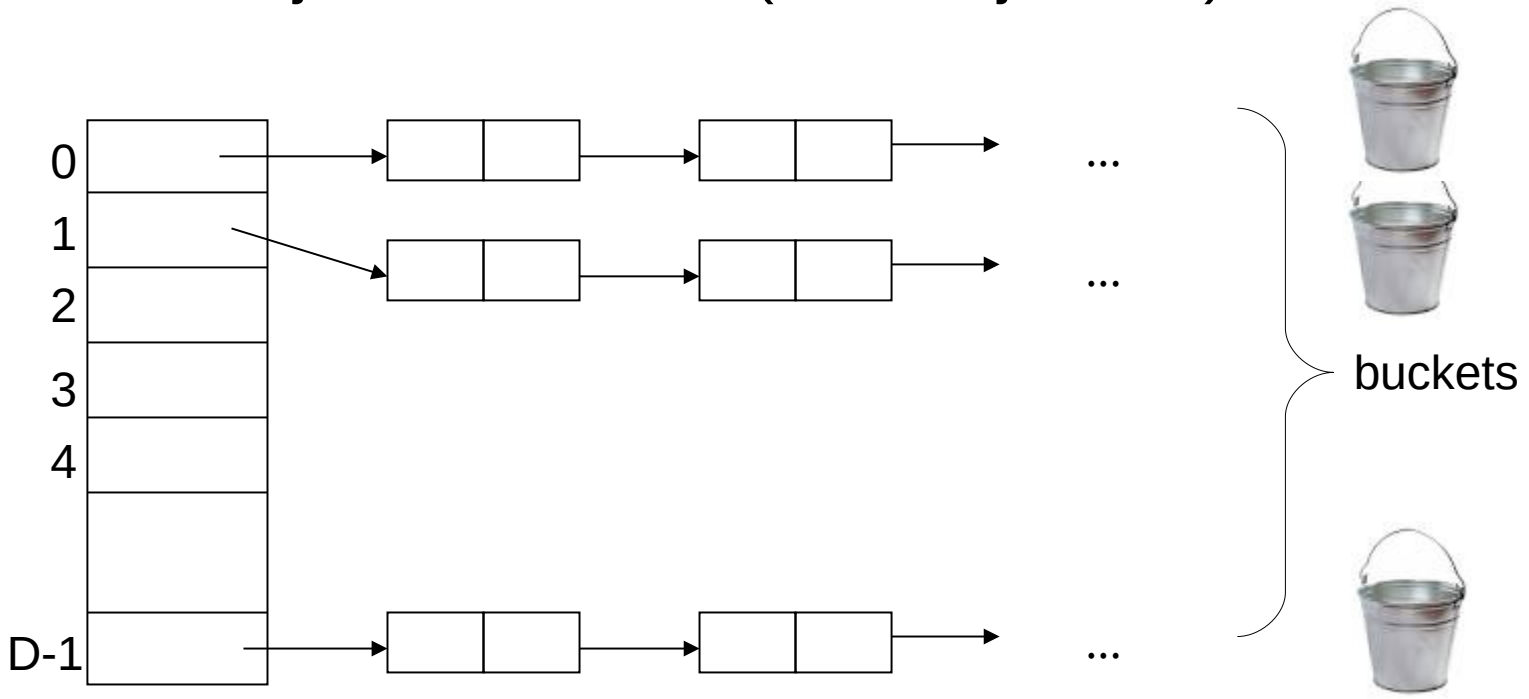
“Pigeon hole principle”

Rješavanje kolizije – nastavak

- Generalno, postoje dva pristupa rješavanju ovog problema:
  - otvoreno hashiranje (open hashing)
  - zatvoreno hashiranje (closed hashing)
- Razlika u ova dva pristupa je u tome gdje se smještaju objekti za čije ključeve se dobija ista hash vrijednost.
- Open hashing – smještaju se izvan hash tabele (niza)
- Closed hashing – smještaju se unutar hash tabele ali u nekoj drugoj ćeliji.
- Rehashing: ponovna izgradnja hash tabele u slučaju porasta broja elemenata u kontejneru.

Rješavanje kolizije – open hashing

- U ovu svrhu se koristi tzv. ulančavanje (chaining).
- Za svaku ćeliju hash tabele kreira se linkana lista objekata (zapisa) čiji ključevi imaju istu hash vrijednost. Na taj način se rješava kolizija.
- Kad se zapis dodaje u ovakvu strukturu podataka računa se hash njegovog ključa te se taj zapis dodaje u odgovarajuću linkanu listu koja je asocirana sa dobijenom adresom (hash vrijednost).



Chaining – analiza

- Ako je broj ključeva N u tabeli mnogo veći od D (veličina tabele, tj. broj linkanih listi), aproksimacija prosječne veličine linkane liste asocirane sa svakom ćelijom tabele je N/D jer svaka od D hash vrijednosti ima istu vjerovatnoću pojavljivanja zbog samog dizajna hash funkcije.
- D se bira relativno malo da ne bismo zauzimali previše memorije za pokazivače na linkane liste, kao i da pretraga u listi bude brža..
- Obično se za D uzima oko N/10 kako bi se dobile relativno kratke liste (oko 10 elemenata) koje se onda pretražuju sekvencijalno.
- Ovaj pristup je najprikladniji kad se hash tabela čuva u internoj memoriji.
- Load factor (prosječan broj poređenja): $\alpha = N/D$
- Time se dobija približno konstantno vrijeme pretraživanja $O(1+\alpha)$.

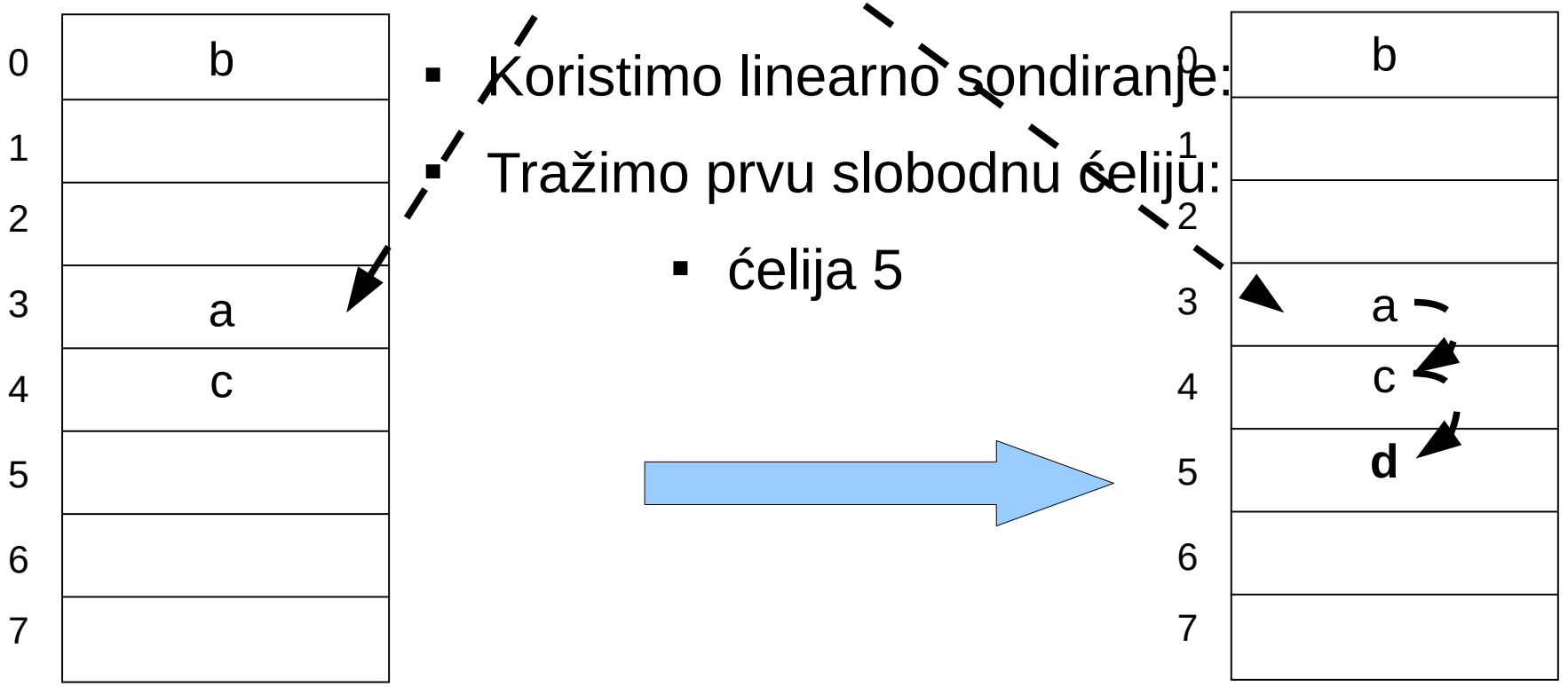
Rješavanje kolizije – closed hashing

- Za ovaj pristup su asocirana dva metoda:
  - linearno sondiranje (linear probing)
  - dvostruko hashiranje (double hashing)
- Oba pristupa se temelje na pronalasku slobodne ćelije unutar hash tabele gdje će se smjestiti podatak.
- Kod linearnog sondiranja, u slučaju zauzeća ćelije, traži se prva sljedeća slobodna ćelija. Inkrement je obično 1 ali može biti i neki drugi broj: h(x) = (x+i) % D
- Princip je sličan i kod dvostrukog hashiranja. Razlika je u tome što se inkrement određuje drugom hash funkcijom: h(x) = (x + h2(x)) % D
  - treba paziti da h2(x) ne daje 0 jer dolazi do beskonačnog ponavljanja kolizija.
  - D i h2 moraju biti međusobno prosti jer u protivnom brzo dolazi do ponavljanja sekvenci sondiranja.

Rješavanje kolizije – closed hashing

- Neka je D=8, i ključevi *a,b,c,d* imaju hash vrijednosti:

  h(a)=3, h(b)=0, h(c)=4, h(d)=3
- Gdje ćemo ubaciti **d**? Ćelija 3 je već zauzeta.



- Koristimo linearno sondiranje:
- Tražimo prvu slobodnu ćeliju:
  - ćelija 5

Linearno sondiranje i dvostruko haširanje – analiza

- Kod ovih pristupa broj ćelija hash tabele mora biti veći od broja ključeva.

Linearno sondiranje

- U slučaju kolizije, linearno sondiranje u prosjeku koristi manje od 5 provjera za hash tabelu koja je manje od 2/3 puna.
- Kako se hash tabela popunjava, kod približno punih tabela nastaje veći broj kolizija što dovodi do popunjavanja veće kontinualne sekvence ćelija i usporavanja performansi. Ova pojava se naziva ***clustering***.

Dvostruko sondiranje

- Dvostruko sondiranje rješava problem clusteringa i u prosjeku daje bolje rezultate od linearnog sondiranja.

U oba slučaja imamo problem kod brisanja podataka (nastaju praznine).

Primjena hash tabela

- Baze podataka: hash tabele pružaju način za lociranje podataka u konstantnom vremenu.
- Kompajleri: tabele simbola.
- Imenici podataka (*dictionary*): strukture podataka koje omogućavaju dodavanje, brisanje i pretraživanje podataka. Mogu se implementirati i drugim strukturama podataka ali korištenje hash tabela je naročito efikasno.
- Mrežni aloritmi: rutiranje, klasifikacija paketa, nadgledanje mreže.
- Browser cache

- Standardna biblioteka:
  - Header: `<unordered_map>`
  - Tip: std::unordered_map`<TypeKey, TypeVal>`

Za računanje hash funkcije defaultno se koristi objekat tipa: **`std::hash<TypeKey>`**, `tzv. hasher`

Primjena hash tabela – hasher

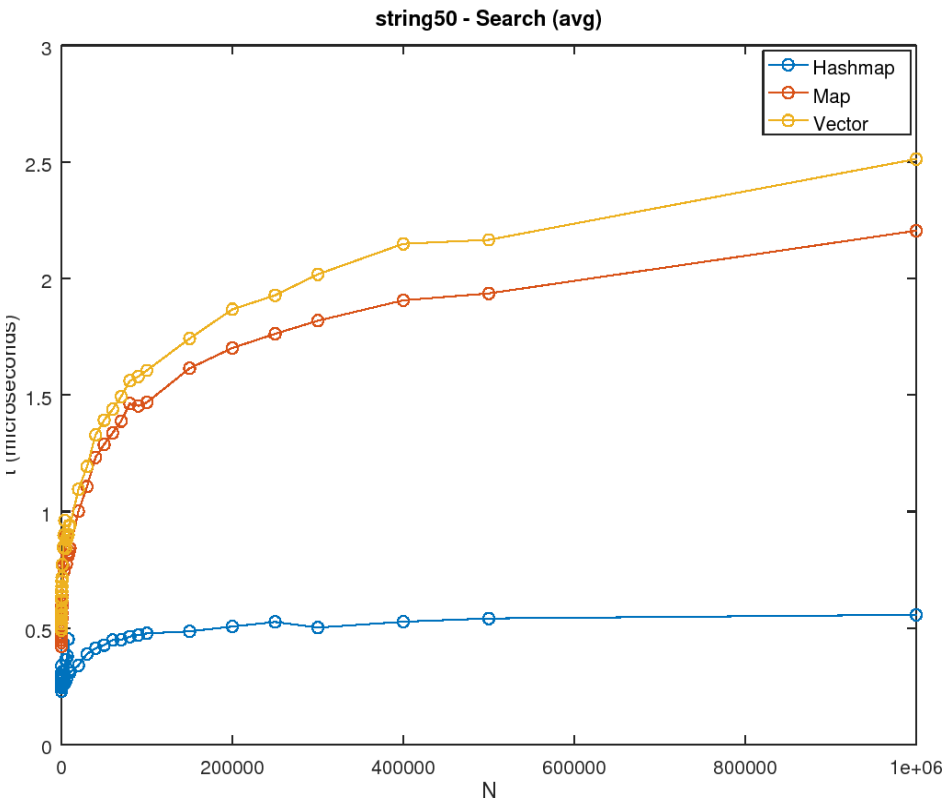
```
struct A {
  int x;
};

std::unordered_map<A, ValueType> ht;
```

```
#include <functional>
namespace std {
  template <>
    struct hash<A> {
      std::size_t operator()(const A& k) const {
        return hash<int>{}(k.x);
      }
    };
}
```

- Usporedba performansi za std: map, vector i unorder_map

Pretraga objekta koji za čuvanje podataka **koristi** dinamičku alokaciju (std::string)



Pretraga objekta koji za čuvanje podataka **ne koristi** dinamičku alokaciju