# Algoritmi in podatkovne strukture – 2

## Slovar

osnove, izvedba s seznamom

# Slovar

Imamo slovar $S$ nekakšnih elementov. S tem slovarjem želimo početi vsaj naslednje operacije:

**dodajanje:** `Insert(`$S$`, x)` – v slovar $S$ dodamo nov element `x`.

**iskanje:** `Find(`$S$`, x) --> y` – v slovarju $S$ poiščemo element `x`. Rezultat `y` je lahko Boolova vrednost `true` ali `false`, ali pa neki podatki povezani z elementom `x`.

**izločanje:** `Delete(`$S$`, x) --> y` – iz slovarja $S$ izločimo element `x`. Rezultat `y` je lahko Boolova vrednost `true` ali `false`, ki sporoči ali je bil element uspešno izločen ali ne, ali pa operacija ničesar ne vrne.

Slovar je v resnici podoben matematični strukturi množica, le da pri slovarju elementi lahko sestoje še iz podatkov (glej `y` pri iskanju).

# Primer

Imejmo slovar, v katerem so elementi: $S = \{45, 2, 17, 46\}$. Potem lahko sledimo operacijam nad njim.

*Pozor: Ničesar nismo rekli o tem, kako so* ORGANIZIRANI *(shranjeni, strukturirani) elementi v slovarju $S$!*

# Slovar kot APS oziroma predmet

Slovar lahko obravnavamo kot *abstraktno podatkovno strukturo*. V tem primeru definiramo operacije kot funkcije nad podatkovno strukturo slovar $S$.

Lahko pa obravnavamo slovar kot *predmet* (predmetno naravnana tehnologija) in v tem primeru postanejo operacije nad slovarjem njegove metode.

Pri tem predmetu (domače naloge) boste vse podatkovne strukture naredili kot predmete (definirali boste ustrezne razrede). Prevedba v APS je preprosta. Uporabimo jo pri ne-predmetno naravnanih programskih jezikih (npr. C, Pascal, FORTRAN itd.).

# Oblika elementov v slovarju

Elementi, ki so shranjeni v slovarju imajo lahko naslednje lastnosti:

- lahko so samostojni
- lahko sestoje iz ključa in podatkov:

```
public class Elt {
  public Object key;
  public Object data;
}
```

Pri tem predmetu bomo predpostavili, da so elementi slovarja vedno pari (prim. z y na prosojnici 3).

# Oblika ključev elementov v slovarju

- lahko so kakršnekoli oblike – `Object`
- lahko so takšne oblike, da jih lahko primerjamo ali so enaki ali ne
- lahko jih primerjamo po velikosti

Pri tem predmetu bomo predpostavili, da so elementi slovarja vedno cela števila, da poenostavimo razlago. Sicer bi morali podrobneje definirati razred `Elt` na prosojnici 6.

DOMAČA NALOGA: definirajte `Elt` podrobneje. Uporabite namesto definicije razreda, vmesnik. V jeziku Java v standardnih knjižnicah že obstajajo podobni razredi – poiščite jih.

Vse odgovore objavite v forumu!

# Elt

```
public class Elt {
  public int key;
  public Object data;
}
```

# slovar

```
public interface Slovar {
  public void Insert(Elt x);
  public Object Find(int key);
    // če elementa ni v slovarju vrne NULL, sicer podatke
  public Elt Delete(int key x);
}
```

# Še enkrat slovar

Za algoritme in podatkovne strukture nas zanima, da so:

- pravilni in
- kako učinkoviti so.

Pri vajah in domačih nalogah bomo opazovali predvsem zadnjo lastnost. Zato, bomo vsem podatkovnim strukturam dodajali lastnosti in metode, ki nam bodo omogočale opazovanje učinkovitosti.

# Slovar, ki ga lahko opazujemo

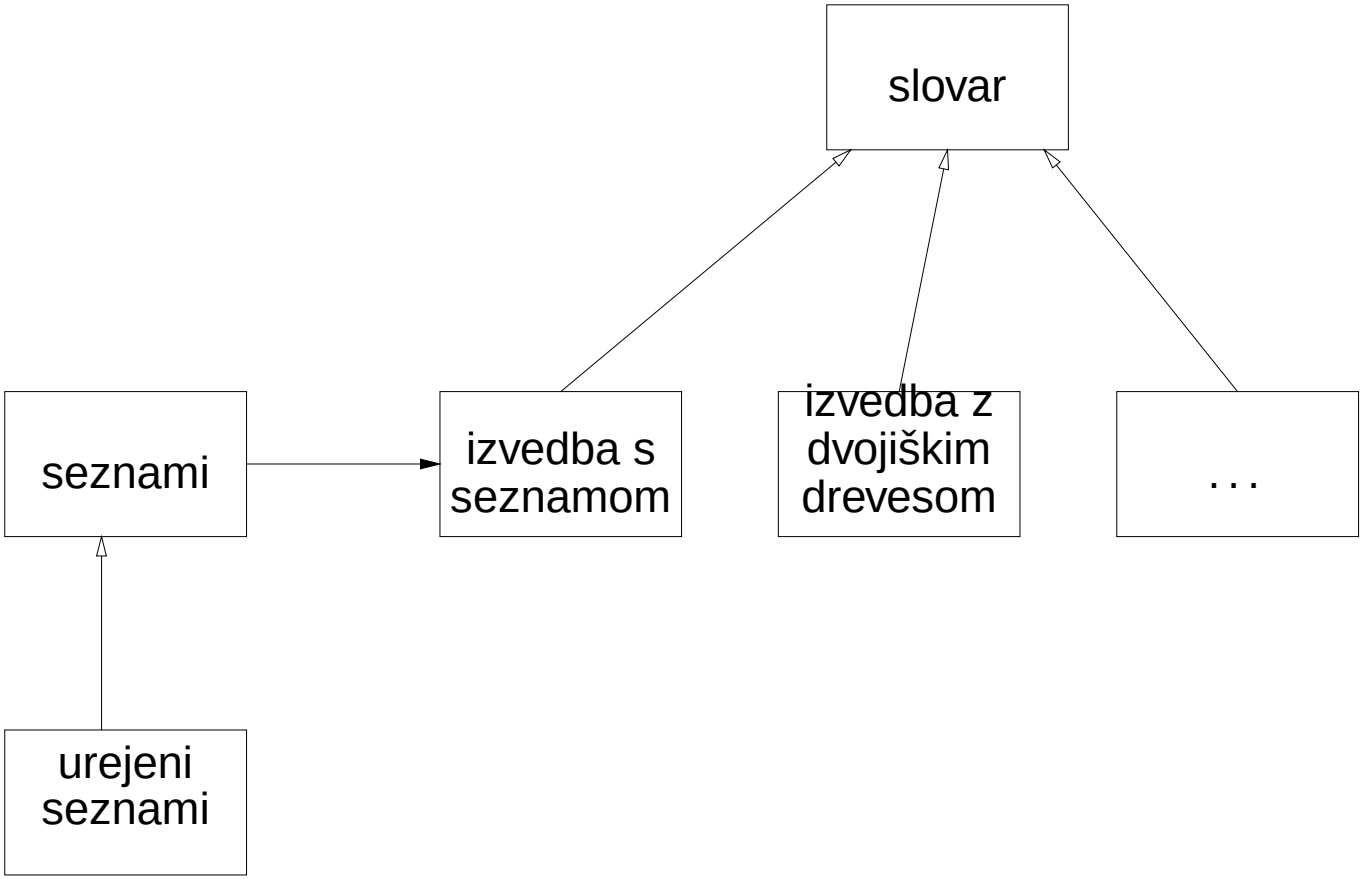
```
public interface Slovar {
  public void Insert(Elt x);
  public Object Find(int key);
    // če elementa ni v slovarju vrne NULL, sicer podatke
  public Elt Delete(int key x);
  // -----------
  public int Dostopov();
    // vrne število dostopov do pomnilnika (elementov
    // podatkovne strukture
  ...
}
```

# Izvedba slovarja

Ogledali si bomo več izvedb slovarja. Nekaj smo jih že srečali, nekaj pa bo povsem novih:

- izvedba s seznamom: neurejen, urejen
- izvedba z dvojiškimi drevesi: iskalna, uravnotežena, AVL, rdeče-črna drevesa
- izvedba z večsmernimi drevesi: B-drevesa, 2-3 drevesa
- izvedba z razpršenimi tabelami: odprto naslavljanje in veriženje
- izvedba s preskočni seznami

# Izvedba slovarja



# Seznami

Recimo, da imamo seznam $[45, 2, 17, 46]$, potem velja:

- da je $45$ glava seznama in $[2, 17, 46]$ rep;
- da je $2$ glava seznama in $[17, 46]$ rep;
- da je $17$ glava seznama in $[46]$ rep;
- da je $46$ glava seznama in $[]$ rep;

Seznam sestoji iz:

- glave, ki bo v našem primeru iz razreda `Elt` in
- repa, ki je ponovno seznam

# Razred Seznam

```
public class Seznam {
  private Elt glava;
  private Seznam rep;
  public Seznam Insert(Elt element) {
    return new Seznam(element, this);
  }
  public Object Find(int key) {
    if (glava.key == key) return glava.data;
    else if (rep == NULL) return NULL;
    else return rep.Find(key);
  }
  public nekajDrugega Delete(int key) {
    // za domačo nalogo
  }
}
```

# Brisanje elementa iz seznama

Pri brisanju moramo vrniti dva predmeta:

- sam element, ki smo ga izbrisali (glavo seznama, ki smo jo izbrisali) in
- rep seznama, ki je ostal, saj le-ta predstavlja nov seznam, v katerem ni več brisanega elementa

Zato moramo definirati nov razred `nekajDrugega` (dajte mu smiselnejše ime), ki bo vseboval oba predmeta in bo predstavljal rezultat metode `Delete`.

# Sledenje učinkovitosti

Učinkovitosti bomo sledili tako, da bomo šteli **število dostopov** do podatkovne strukture.

VPRAŠANJE: kakšna je povezava med številom dostopov in zahtevnostjo algoritma?

To lahko izvedemo na različne načine. Najpreprosteje je morda tako, da definiramo nov razred `Stevec`, predmet katerega lahko vedno uporabimo za štetje dostopov (in še česa).

Novi razred moramo moči pogledati, povečati (za 1) in nastaviti na začetno vrednost.

# Razred `Counter`

```
public class Counter {
  private int value;
  public Counter()   { value= 0     };
  public void Inc()  { value++      };
  public int Value() { return value };
  public void Set()  { value= 0     };
}
```

# Sledenje v seznamu

```
public class Seznam {
  private Elt glava;
  private Seznam rep;
  public Seznam Insert(Elt element) {
    return new Seznam(element, this);
  }
  public Object Find(int key, Counter cnt) {
    cnt.Inc();  // pri dostopu do glave seznama povečamo števec za 1
    if (glava.key == key) return glava.data;
    else if (rep == NULL) return NULL;
    else return rep.Find(key, cnt);
  }
  public nekajDrugega Delete(int key, Counter cnt) {
    // za domačo nalogo
  }
}
```

DOMAČA NALOGA: Zakaj to deluje? Zakaj lahko števec `cnt` damo metodi kot parameter in bomo še vedno zunaj metode opazili spremembo? (*Namig:* prenos parametrov.)

# Seznam in slovar

Za izvedbo operacij slovarja uporabimo seznam – naše podatke v slovarju organiziramo (*strukturiramo*) kot seznam.

```
public class SeznSlovar implements Slovar {
  private Seznam  dictionary;
  private Counter cnt;
  public SeznSlovar() {
    dictionary= NULL;
    cnt= new Counter();
  }
  public void Insert(Elt x) {
    if (dictionary == NULL) dictionary= new Seznam(x);
    else dictionary= dictionary.Insert(elt);
  }
```

# Seznam in slovar – nadaljevanje

```
  public Object Find(int key) {
    if (dictionary == NULL) return NULL;
    else return dictionary.Find(elt, cnt);
  }
  public Elt Delete(int key x) { ... }
  // -----------
  public int Dostopov() { return cnt.Value() }
  ...
}
```

# Opomba za v naprej

Odslej bomo v glavnem zanemarjali uporabo števca, saj bo očitna iz konteksta.

Tudi koda bo še bolj podana v koščkih, da se izognemu preveliko informacijam na prosojnici.

V glavnem vse boste naredili bodisi na vajah bodisi pri domačih nalogah.

# Urejeni seznam

Namesto navadnega seznama lahko uporabimo urejen seznam, v katerem so elementi urejeni po velikosti (glej lastnosti na prosojnici 7).

Ničesar se ne spremeni v razredu `SeznaSlovar`, le namesto razreda `Seznam` uporabimo razred `UrejSeznam`.

# Urejeni seznam

```
public class UrejSeznam extends Seznam {
  private Elt glava;
  private UrejSeznam rep;
  public Seznam Insert(Elt element) {
    return new UrejSeznam(element, this);
  }
  public Object Find(int key, Counter cnt) {
    cnt.Inc();
    if (glava.key == key) return glava.data;
    if (glava.key < key)  return NULL;
    else if (rep == NULL) return NULL;
    else return rep.Find(key, cnt);
  }
  public nekajDrugega Delete(int key, Counter cnt) {
    // za domačo nalogo
  }
}
```

# Zapletenost

| | `Find` | `Insert` | `Delete` |
|---|---|---|---|
| seznam | $O(n)$ | $O(1)$ | $O(n)$ |
| urejen seznam | $O(n)$ | $O(n)$ | $O(n)$ |

- Kakšna je velikost?
- Kaj pa najboljši čas?
- In kaj pa povprečen čas?
- So zgornje vrednosti smiselne?
- Recimo, da poznamo vsa poizvedovanja vnaprej – optimalni čas.
- MTF (*move to front*) seznam.
- KISS – *Keep it simple, stupid.*

# Inženirska vprašanja

- Ali je `UrejSeznam` v resnici razširitev razreda `Seznam`? Če ne, kaj je potrebno popraviti?
- Ali je diagram s prosojnice 13 pravilen?
- Nerekurzivne rešitve?

Odgovore objavite na forumu.