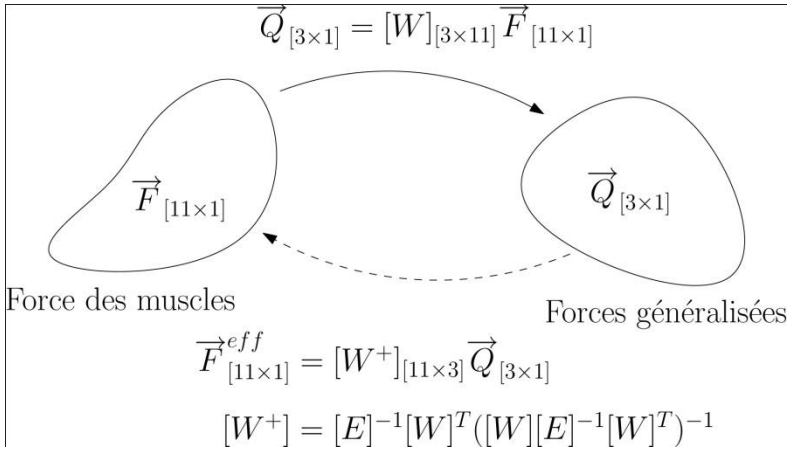


ADT PRESLIKAVA (angl. *map*)

ADT preslikava

Preslikava vsakemu elementu **d** iz domene priredi ustrezen element **r** iz zaloge vrednosti:

$$M(d) = r$$

Primer: Preslikava *M(d)=r* vsaki telefonski številki *d* priredi lastnika računa *r*

Telefonska številka		Lastnik računa
031 - 456 876	→	Janez Novak
031 – 345 987	→	Miha Kolar
041 – 237442	→	Tine Boh
040 – 327 896	→	Ana Hozjan
070 – 213 445	→	Tina Zaletel

ADT preslikava

Operacije definirane za **ADT MAPPING**:

- **`MAKENULL(M)`** – `inicializira prazno preslikavo`
- **`ASSIGN(M, d, r)`** – `definira, da je M(d) = r`
- **`COMPUTE(M, d)`** – `vrne vrednost M(d), če je definirana, sicer` **`null`**

Pri implementaciji v javi lahko definiramo vmesnik:

```
public interface Mapping {
   public abstract void makenull() ;
   public abstract void assign(Object d, Object r) ;
   public abstract Object compute(Object d) ;
}
```

ADT preslikava

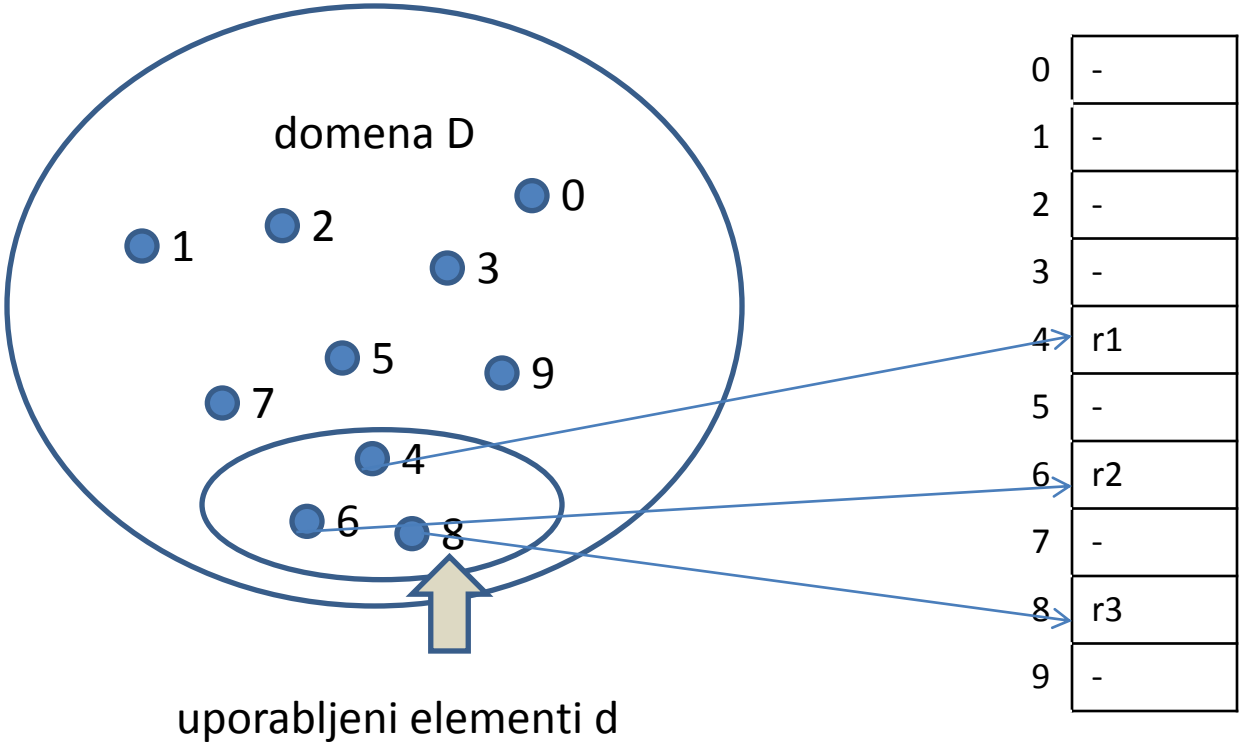
Pri implementaciji preslikave lahko uporabimo:

- **seznam** parov (d, r) – časovna zahtevnost iskanja parov je reda O(n), kar je pogosto nesprejemljivo.

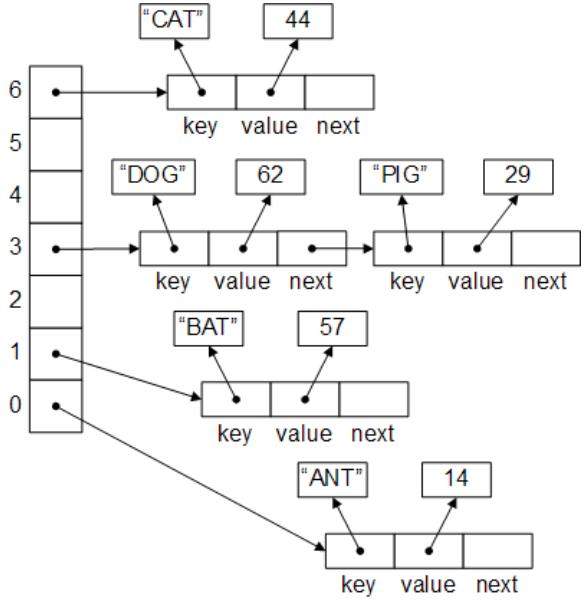
- **iskalna drevesa** – časovna zahtevnost iskanja parov je reda O(log n), kar je v nekaterih aplikacijah sprejemljivo.

- **polje**
  - indeksi polja predstavljajo elemente domene d
  - hitro dodajanje, iskanje, brisanje - O(1)
  - zahteva veliko pomnilnika
  - domena mora biti zadosti majhna

Implementacija preslikave s poljem



- časovna zahtevnost iskanja in dodajanja elementa je reda O(1)
- taka implementacija je možna samo, če je moč domene zadosti majhna (velikost polja je enaka moči domene)



ZGOŠČENA TABELA (angl. *hash table*)

Zgoščena tabela

- v praksi je moč domene velika ali celo neskončna
- uporabimo **zgoščevalno funkcijo** (hash function), ki preslika (zgosti) originalno domeno v manjšo domeno

$$h : domaintype \rightarrow smalldomaintype$$

- zgoščevalna funkcija “razprši” elemente po manjši domeni. Podatkovni strukturi, ki uporablja zgoščevalno funkcijo, pravimo zgoščena tabela (hash table)

- želimo funkcijo, ki čim bolj enakomerno “razprši” elemente po polju

Primer zgoščene tabele

Lahko uporabimo zgoščevalno funkcijo:

```
h(tel_st) = tel_st mod 17
```

Telefonska številka		Lastnik računa
031 - 456 876	→	Janez N.
031 – 345 987	→	Miha K.
041 – 237442	→	Tine B.
040 – 327 896	→	Ana H.
070 – 213 445	→	Tina Z.
040 – 743 210	→	Marko S.
070 – 236 580	→	Teja B.

h(031456876) = 8
h(031345987) = 10
h(041237442) = 15
h(040327896) = 3
h(070213445) = 11
h(040743210) = 7
h(070236580) = 9

0	-
…	-
3	Ana H.
…	-
7	Marko S.
8	Janez N.
9	Teja B.
10	Miha K.
11	Tina Z.
…	-
15	Tine B.
16	-

z uporabo zgoščevalne funkcije preslikamo indekse iz originalne domene v indekse manjše domene

Zgoščena tabela

PROBLEMI:

- Izbira velikosti polja (tabele):
  - zadosti velika, da lahko spravimo vanjo vse elemente
  - ne prevelika, saj sicer zaseda preveč pomnilnika

- Izbira ustrezne zgoščevalne funkcije
  - želimo funkcijo, ki enakomerno "razprši" elemente
  - Najbolj preprosta: $h(x) = x \bmod m$
  - m - praštevilo, ki se razlikuje od potence 2^i
  - Idealno: injektivna

$$d_1 \neq d_2 \Rightarrow h(d_1) \neq h(d_2)$$

- Sovpadanje:

$$d_1 \neq d_2 \wedge h(d_1) = h(d_2)$$

Zgoščena tabela: reševanje problemov

OMEJENA VELIKOST tabele:

PONOVNO ZGOŠČANJE (rehashing)

Ko se tabela napolni, zgradimo večjo tabelo ter vse elemente iz manjše tabele uvrstimo v novo tabelo z novo zgoščevalno funkcijo.

Ponovno zgoščanje (rehashing) zahteva *O(n)* časa.

V povprečju: za *n* elementov najmanj *n* vstavljanj:
n operacij *O(1)* in *1* operacija *O(n)* → *v povprečju* $2n/(n+1) = O(2) = O(1)$

Zgoščena tabela: reševanje problemov

SOVPADANJE: v praksi je neizogibno

- z **zaprto zgoščeno tabelo** in z zaporednim naslavljanjem: uporabljamo zaporedje zgoščevalnih funkcij – v primeru sovpadanja izračunane vrednosti uporabimo drugo funkcijo za naslednji možni položaj elementa

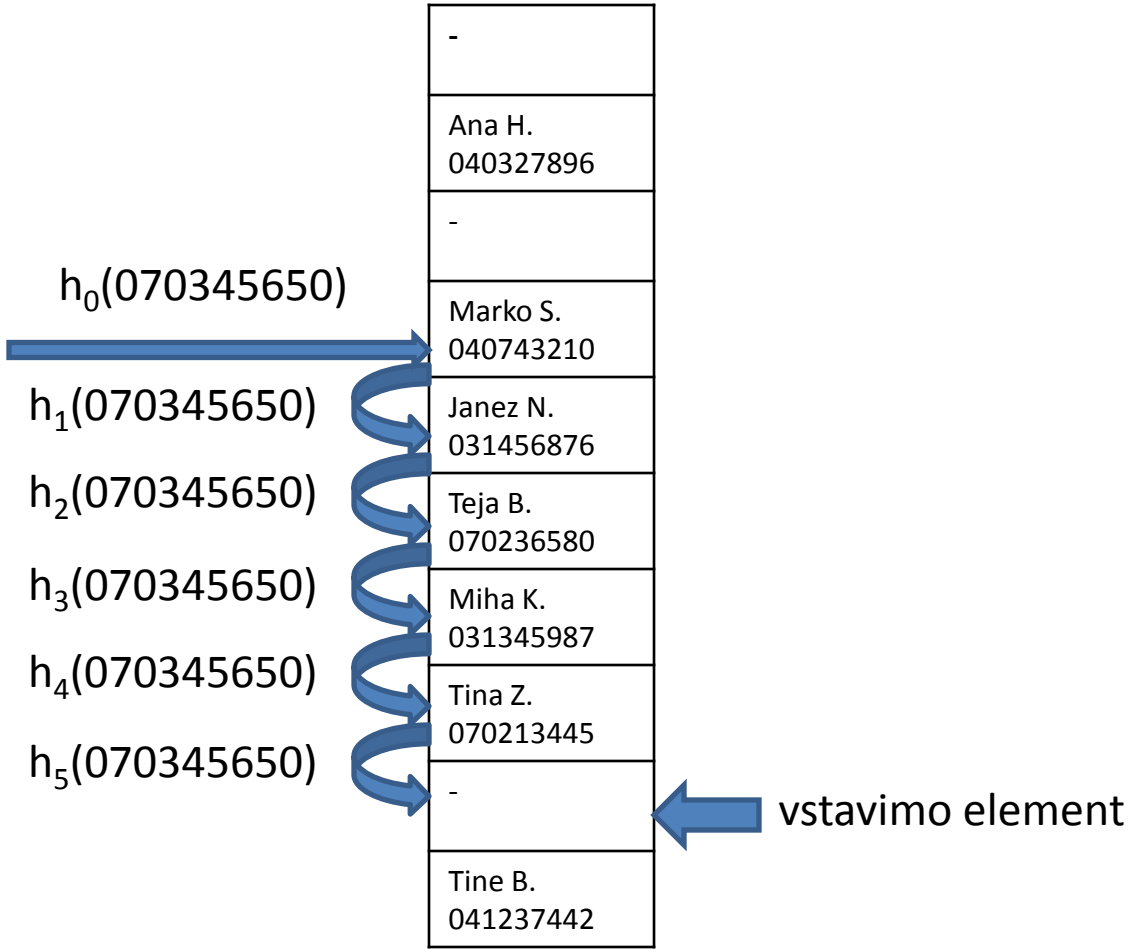
Najpreprostejše zaporedje: $h_i'(x) = (h(x) + i) \bmod m$

Zaprta zgoščena tabela

Primer preprostega zaporedja zgoščevalnih funkcij:

$$h_i'(x) = (h(x) + i) \bmod m$$

Telefonska številka	Lastnik računa
070 - 345 650 →	Tomaž I.



Zgoščena tabela: reševanje problemov

SOVPADANJE: v praksi je neizogibno

- z **zaprto zgoščeno tabelo** in z zaporednim naslavljanjem: uporabljamo zaporedje zgoščevalnih funkcij – v primeru sovpadanja izračunane vrednosti uporabimo drugo funkcijo za naslednji možni položaj elementa

Najpreprostejše zaporedje: $h_i'(x) = (h(x) + i) \bmod m$

Boljše: $h_i'(x) = ((h_1(x) + i \times h_2(x)) \ mod \ m)$

$h_1(x) = (x \ mod \ m)$

$h_2(x) = (x \ mod \ m')$ $\qquad m' = m - 2$

- z **odprto zgoščeno tabelo**: v polju hranimo kazalce na sezname parov, ki sovpadajo

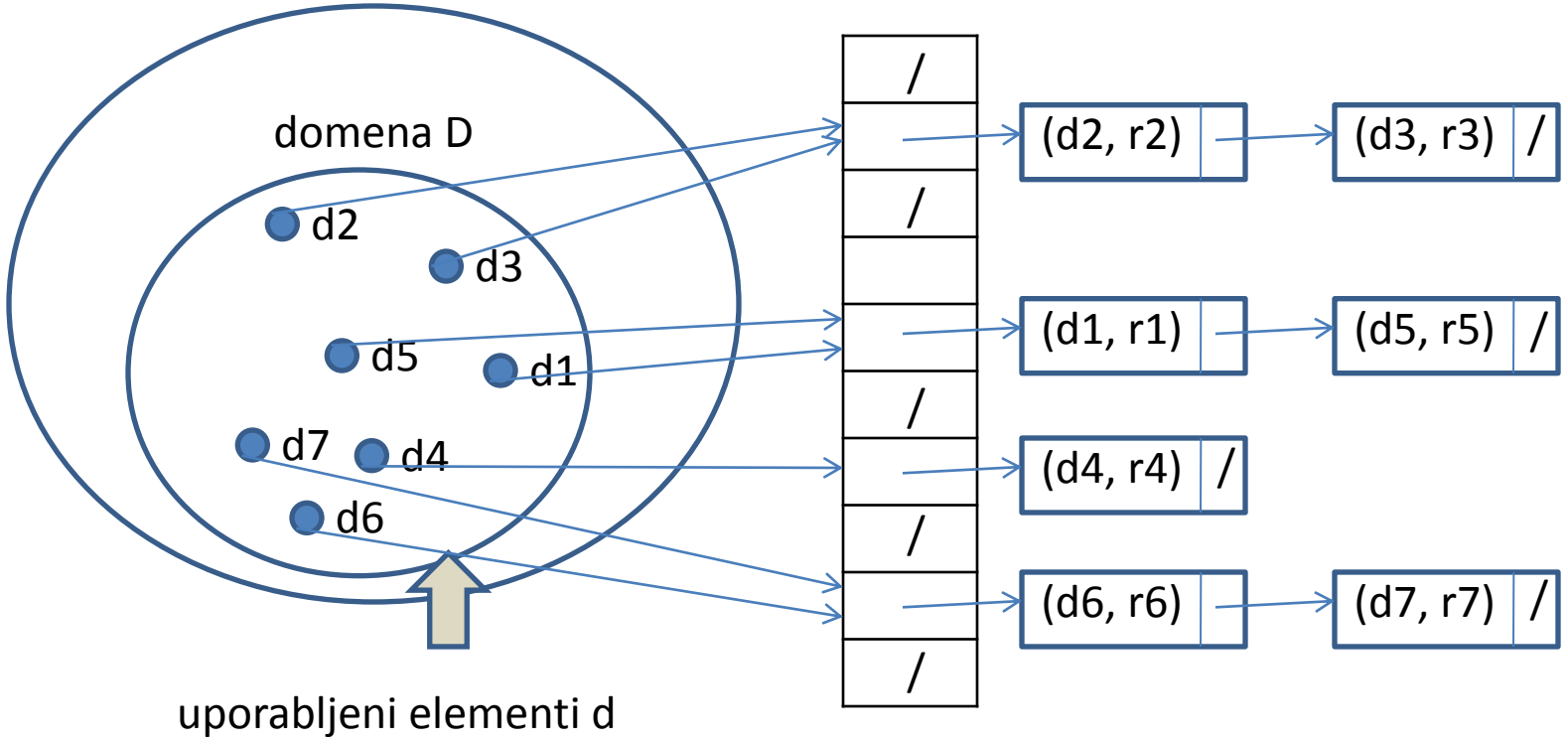
Zaprta zgoščena tabela

Slabosti:

- Pri iskanju elementa je potrebno preiskovati do prvega praznega prostora
- Pri brisanju elementa je treba zapisati posebno vrednost, ki ne zaključi iskanja
- Velikost tabele mora biti večja od števila elementov

  $m > n$
- Če se *n* približa *m,* postanejo operacije nesprejemljivo počasne.

Odprta zgoščena tabela



Pričakovana zahtevnost iskanja elementa je *O(n/m).*

n - število vstavljenih elementov
m - velikost zgoščene tabele.

Prednosti zgoščenih tabel

✓ Če imamo dobro izbrano velikost tabele
in zgoščevalno funkcijo, **so vse operacije O(1)**

ADT MAPPING:

- **`MAKENULL(M)`** – `O(1)`
- **`ASSIGN(M, d, r)`** – `O(1)`
- **`COMPUTE(M, d)`** – `O(1)`

✓ **Zaprta** zgoščena tabela zaseda manj pomnilnika
✓**Odprta** zgoščena tabela je bolj dinamična/fleksibilna
✓vstavljanje v **odprto** zgoščeno tabelo vedno *O(1)*

Slabosti zgoščenih tabel

- zaradi fiksne zgoščevalne funkcije zgoščena tabela ne more biti dinamična struktura
  - velikost tabele moramo vnaprej definirati
  - če je tabela prevelika, zapravljamo pomnilnik
  - če je tabela premajhna, prihaja do prevelikega sovpadanja elementov
  - ponovno zgoščanje vnese počasno operacijo *O(n)*
- najslabši možni primer: vsi elementi sovpadajo → seznam
- ne moremo učinkovito implementirati operacij, ki temeljijo na urejenosti elementov po ključih
  (iskanje najmanjšega/največjega elementa, iskanje naslednjega elementa po velikosti, izpis elementov v danem intervalu vrednosti ključa …)

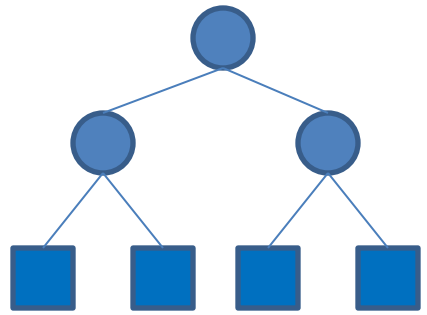
Če so slabosti nesprejemljive → DREVO

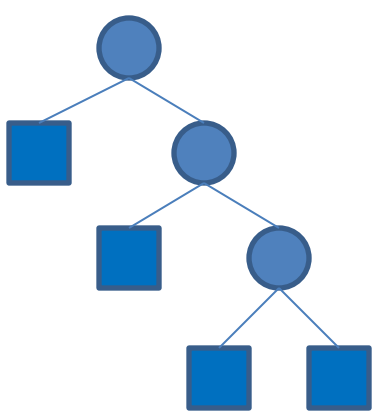
Bistvo drevesa:

- ✓ če je (delno) poravnano: višina O(log n)
- ✓ operacije iskanja, dodajanja in brisanja O(log n)
- ✓ operacije glede urejenosti elementov (min, max, next): O(log n)

Če pa drevo ne bi bilo delno poravnano (ampak izrojeno) → seznam

O(log n)





O(n)

ZBIRKE

(angl. *collections*)

Javanske zbirke

Zbirka (collection) je naziv za skupino podatkov organiziranih v enotni objekt. Ogrodje zbirk (collection framework) ponuja poenoten pogled na zbirke in osnovne metode za njihovo obdelavo:

- določa **VMESNIKE**, ki opisujejo vse metode za delo z zbirko (brez dejanske izvedbe)
- vsebuje učinkovite **IMPLEMENTACIJE** določenih podatkovnih struktur
- vsebuje statične **POMOŽNE METODE** za pogosto rabljene postopke dela z zbirkami
  (na primer sortiranje, mešanje, iskanje in podobno)

Metode vseh zbirk

```
public interface Collection
{
        // osnovne operacije
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(Object element);          // opcijsko
        boolean remove(Object element);       // opcijsko
        Iterator iterator();

        // množične operacije
        boolean containsAll(Collection c);
        boolean addAll(Collection c);         // opcijsko
        boolean removeAll(Collection c);      // opcijsko
        boolean retainAll(Collection c);      // opcijsko
        void clear();                         // opcijsko

        // pretvorba v polje
        Object[] toArray();
        Object[] toArray(Object a[]);
}
```

Osnovne zbirke

		implementacija			
		zgoščena tabela	polje	povezan seznam	uravnoteženo drevo
vmesnik	Seznam		★	★	
	Množica	★			★
	Preslikava	★			★

Iteratorji

Iteratorji omogočajo sprehod prek vseh elementov zbirke. Vmesnik:

```
Interface Iterator {
    boolean hasNext(); // preveri, ali je na voljo še kak element
    Object next(); // vrne element ter se prestavi za mesto naprej
    void remove(); // odstrani element
}
```

Primer:

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<String>();
    list.add("Primer");
    list.add("elementov");
    list.add("seznama");

    Iterator<String> itr = list.iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }
}
```

Seznami

Implementacija **s poljem** (**ArrayList**) in **s kazalci** (**LinkedList**).

POLJE: seznam se malo spreminja, pogosto dostop preko indeksov

KAZALCI: seznam se pogosto spreminja, redko dostop preko indeksov

Pomembnejše podedovane funkcije od vmesnika **Collection**:

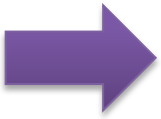
- `add(Object)` – doda nov element na konec seznama
- `remove(Object)` – odstrani prvo pojavitev elementa v seznamu

Metode za **dostop do elementov** preko indeksov (pričnejo se z 0):

- `add(int, Object)`
- `get(int)`
- `set(int, Object)`
- `remove(int)`

Funkcije za **iskanje elementov**:

- `indexOf(Object)`
- `lastIndexOf(Object)`



Seznam najpomembnejših operacij za razreda ArrayList in LinkedList se nahaja v učbeniku, str. 125 in 126

Množice

Neurejena je implementirana z **zaprto zgoščeno tabelo s ponovnim zgoščanjem** (**HashSet**)
Urejena množica je implementirana z **rdeče-črnim drevesom** (**TreeSet**).

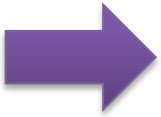
Metode za **dodajanje, iskanje in brisanje elementov**:	HashSet	TreeSet
• `add(Object e)`	`O(1)`	`O(log n)`
• `Contains(Object e)`		
• `remove(Object e)`		

Metodi za **velikost množice**:

- `isEmpty()`
- `size()`

Metode za **operacije glede urejenosti** (samo **TreeSet**):

- `first()` // vrne prvi (min element
- `last()` // vrne zadnji (max) element
- `headSet(Object to)` // vrne podmnožico elementov manjših od to
- `tailSet(Object from)` // vrne podmnožico elementov večjih od from
- `subSet(Object f, Object t)` // vrne podmnožico elementov med f in t



Tabeli najpomembnejših operacij za razreda HashSet in TreeSet se nahajata v učbeniku, strani 127 in 128

Množice

Metode za **množične operacije** omogočajo izvedbo algebre množic:

- **unija**: `s1.addAll(s2);`
- **presek**: `s1.retainAll(s2);`
- **razlika**: `s1.removeAll(s2);`
- **Ugotavljanje podmnožice**: `s1.containsAll(s2);`

Preslikave

Preslikava ***brez urejenosti*** *po ključih* je implementirana z **zaprto zgoščeno tabelo s ponovnim zgoščanjem** (**HashMap**),
Preslikava z ***urejenimi pari*** *po ključih* je implementirana z **rdeče-črnim drevesom** (**TreeMap**).

Metode za **dodajanje, računanje in brisanje parov**:

- `put(Object key, Object value)`
- `remove(Object key)`

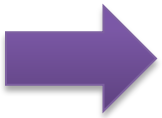
Metoda za **iskanje vrednosti za dani ključ**:

- `get(Object key)`

	HashMap	TreeMap
	O(1)	O(log n)

Metode za poglede na preslikavo skozi zbirke

- `keySet()` // vrne množico vseh ključev preslikave
- `values()` // vrne zbirko vseh vrednosti
- `entrySet()` // vrne množico parov preslikav ključa v vrednost



Tabeli najpomembnejših operacij za razreda HashMap in TreeMap se nahajata v učbeniku, strani 130 in 131

Povzetek

vmesnik	implementacija: zaprta zgoščena tabela, ponovno zgoščanje	implementacija: polje	implementacija: povezan seznam s kazalci	implementacija: rdeče-črno drevo
Seznam		**ArrayList** (primerno, ko se seznami **malo spreminjajo** in potrebujemo **pogoste neposredne dostope** do elementov)	**LinkedList** (primerno, ko se seznami **pogosto spreminjajo** in večinoma **ne potrebujemo direktnega dostopa** do elementov	
Množica	**HashSet** NEUREJENA (primerno, ko potrebujemo **hitre operacije** vstavljanja, brisanja in iskanja elementov)			**TreeSet** UREJENA (primerno, ko potrebujemo operacije vezane na **urejenost elementov**)
Preslikava	**HashMap** NEUREJENA (primerno, ko potrebujemo **hitre operacije** vstavljanja, brisanja in iskanja elementov)			**TreeMap** UREJENA (primerno, ko potrebujemo operacije vezane na **urejenost elementov**)