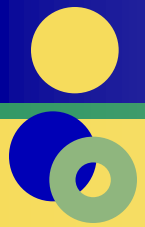
Hash (heš) tabele



Strukture podataka i algoritmi 2

Heš tabela

- Heš tabela je niz u kome se pozicija nekog elementa određuje na osnovu samog elementa

- Funkcija koja izračunava poziciju elementa u nizu se naziva heš funkcija
  - *h*: Object → [0 .. *M* - 1], gde je *M* veličina heš tabele
  - Heš funkcija treba da bude brzo i lako izračunljiva

- Heš tabelama možemo realizovati dve bitne apstraktne strukture podataka: skupove i mape
  - **U odnosu na realizacije skupova/mapa nizovima i listama, heš tabele omogućuju znatno brže pretraživanje**

ADT skup

- Skup je kolekcija objekata koja ne sadrži duplikate

- Operacije u radu sa skupovima
  1. **Dodavanje novog elementa u skup**
  2. **Brisanje elementa iz skupa**
  3. **Pretraživanje – provera da li je element u skupu**
  4. Operacije unije, preseka, razlike

- U praksi su nam često dovoljne samo prve tri operacije
  - Dodavanje elemenata zahteva pretraživanje
  - Brisanje elementa uključuje pretraživanje

ADT mapa

- Termin mapa se u literaturi još naziva i
  - Asocijativni niz (engl. *associative array*)
  - Tabela simbola (engl. *symbol table*)
  - Rečnik (engl. *dictionary*)

- Mapa je kolekcija objekata oblika (ključ, vrednost) pri čemu svaki objekat ima jedinstven ključ
  - **u mapi nema duplikata ključeva**

- Na primer, student je jedinstveno određen brojem indeksa
- Stoga se generacija studenata može predstaviti mapom
  - Ključ – broj indeksa
  - Vrednost – svi ostali podaci o studentu

ADT mapa, operacije

- Dodavanje para (ključ, vrednost) u mapu
  - Prvo proveramo da li mapa već sadrži ključ
  - Dodavanje izvodimo ukoliko mapa ne sadrži dati ključ
- Brisanje para (ključ, vrednost) iz mape
- Pretraživanje (lookup) – dobavljanje vrednost za dati ključ
  - null vrednost ukoliko mapa ne sadrži ključ
- Modifikovanje vrednosti za dati ključ
- Pretraživanje je, slično kao kod skupova, najbitnija operacija

Realizacija ADT-a

- Jedan ADT može biti realizovan (implementiran) na više **fundamentalno** različitih načina
  - Različiti načini za reprezentaciju vrednosti ADT-a
  - Implementacije operacija uslovljene načinom reprezentacije vrednosti ADT-a

- Recimo, skup može biti reprezentovan

  (1) prostim nizom, (2) povezanom listom, (3) heš tabelom

- Različite realizacije ADT-a imaju isti **interfejs** ka korisniku ADT-a
  - Različite klase koje realizuju isti ADT imaju ista zaglavlja metoda koja direktno korespondiraju operacijama ADT-a

Specifikacija ADT-a

- Specifikacija ADT-a je apstraktni deo ADT-a
  - **zaglavlja public** metoda kojima definišemo operacije ADT-a
  - Ono što korisnik ADT-a treba da zna kako bi koristio ADT
  - **Različite realizacije ADT-a dele istu specifikaciju ADT-a**

- U PJ Java ADT-ove možemo specificirati **Java interfejsima**
  - Interfejs se sastoji od definicija konstanti i zaglavlja metoda
  - Java klase implementiraju Java interfejse
  - Ako **ne-apstraktna** klasa *A* implementira interfejs *I* tada *A* mora definisati sva zaglavlja data u *I*
  - Korisniku je dovoljno da zna interfejs da bi koristio klasu
  - Interfejsi će detaljnije biti pokriveni kursom OOP1

Specifikacija ADT skup

```
public interface Set<T> {

    /**
     * Dodaje element u skup.
     * Vraca false ukoliko element vec postoji u skupu
     */
    boolean insert(T element);

    /**
     * Brise element iz skupa.
     * Vraca false ukoliko element ne postoji u skupu.
     */
    boolean remove(T element);

    /**
     * Proverava da li je element u skupu.
     */
    boolean member(T element);
}
```

Specifikacija ADT mapa

```
public interface Map<K, V> {

	/**
	 * Dodaje novi par (key, value) u mapu. Vraca
	 * false ukoliko key vec postoji u mapi.
	 */
	boolean insert(K key, V value);

	/**
	 * Brise par (key, value) iz mape. Vraca
	 * false ukoliko key ne postoji u mapi
	 */
	boolean delete(K key);

	/**
	 * Vraca value vezan za key
	 */
	V get(K key);

	/**
	 * Modifikuje value vezan za key. Vraca
	 * false ukoliko key ne postoji u mapi
	 */
	boolean modify(K key, V value);
}
```

Heš kod

- Heš funkcija mora biti konzistentna
  - $k_1 = k_2 \rightarrow hash(k_1) = hash(k_2)$.
- Svaka Java klasa implicitno nasleđuje klasu Object
- Klasa Object definiše metod
  - ```
    int hashCode()
    // Translate this object to an integer, such that
    // x.equals(y) implies x.hashcode() == y.hashcode().
    ```
- Heš kod nije heš funkcija, ali ćemo heš kod iskoristiti da napravimo haš funkciju u opštem slučaju
    ```

Heš funkcija

- Ako je veličina heš tabele *M* tada heš funkcija treba da vrati broj u opsegu [0 .. *M* – 1]

```
public int hash(Object o, int hashTableSize) {
     return Math.abs(o.hashCode() % hashTableSize);
}
```

- Redefinišući metod hashCode() u našoj klasi dobijamo heš kod za objekte naše klase

- Ti se objekti mogu proslediti u gornju metodu *hash* te tako dobijamo vrednost heš funkcije za naše objekte

Kolizije i princip uniformnosti

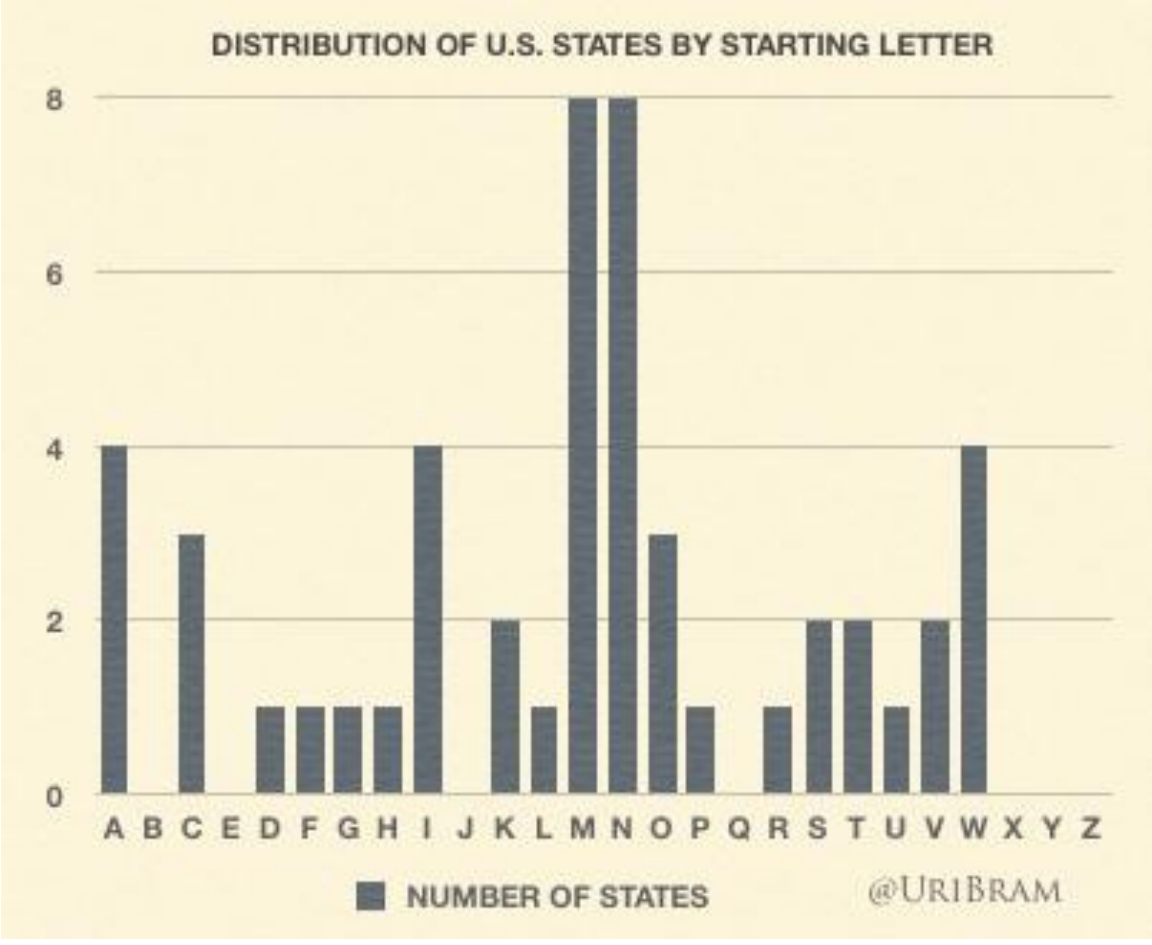
- Dva objekta mogu imati istu vrednost heš funkcije

  $k_1 \neq k_2$ ali $hash(k_1) = hash(k_2)$
- Ovakva situacija se naziva **kolizijom**

- Princip uniformnosti
  - Neka je M veličina heš tabele
  - Neka je O skup objekata takav da je $|O| >> M$
  - Heš funkcija particioniše skup O u familiju skupova H_k, $0 \leq k < M$, gde H_k sadrži objekte sa istom vrednošću heš funkcije
  - Dobra heš funkcija: $|H_0| \approx |H_1| \approx ... \approx |H_{M-2}| \approx |H_{M-1}| \approx |O| / M$

Primer...

- O = skup imena država USA
- M = 26
- hash(s) = s.charAt(0) – ‘A’ **nije uniformna**



Primer 2: hashCode klase String

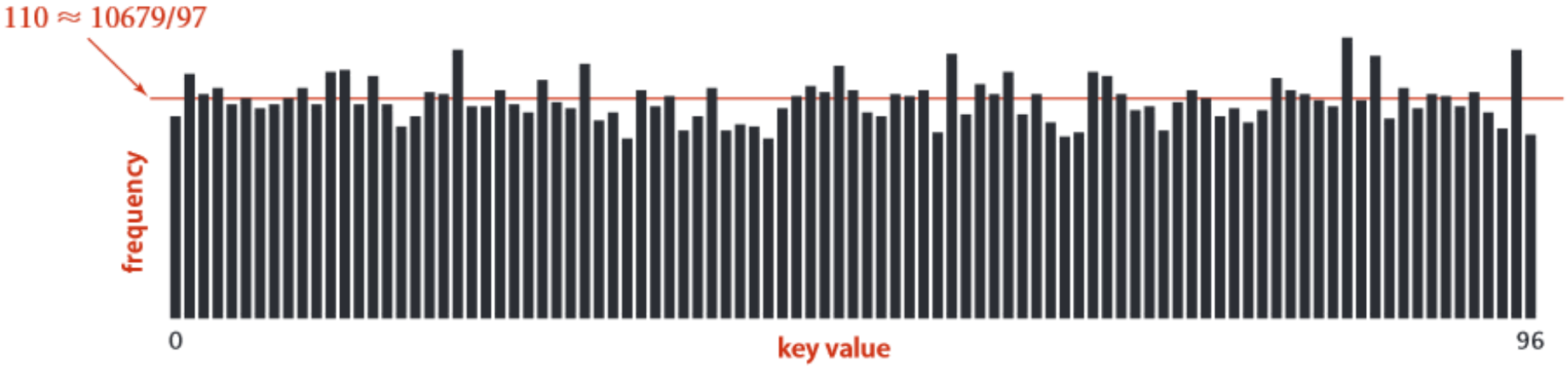
```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

- L = length() - 1
- hash = $s[0] \cdot 31^{L} + s[1] \cdot 31^{L-1} + s[2] \cdot 31^{L-2} + \ldots + s[L-1] \cdot 31 + s[L]$

Primer 2: uniformnost

- O = reči koje se pojavljuju u romanu “Priča o dva grada” Čarlsa Dikensa
  - 10679 različitih reči
- M = 97



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, $M = 97$)

Mehanizmi razrešenja kolizija

- **Otvoreno hešovanje (zatvoreno adresiranje)**
  - Heš tabela je niz jednostruko povezanih listi
  - Liste sadrže objekte sa istom vrednošću heš funkcije
  - Ove liste još zovemo lancima kolizija

- **Zatvoreno hešovanje (otvoreno adresiranje)**
  - Heš tabela je niz, pozicija objekta *o* u nizu je hash(*o*)
  - Ukoliko je pozicija zauzeta tada se određuje nova pozicija po nekoj strategiji
  - Linearno probavanje: hash(*o*) + 1, hash(*o*) + 2, ...
  - Kvadratno probavanje:
    hash(*o*) + 1, hash(o) + 4, hash(o) + 9, ...

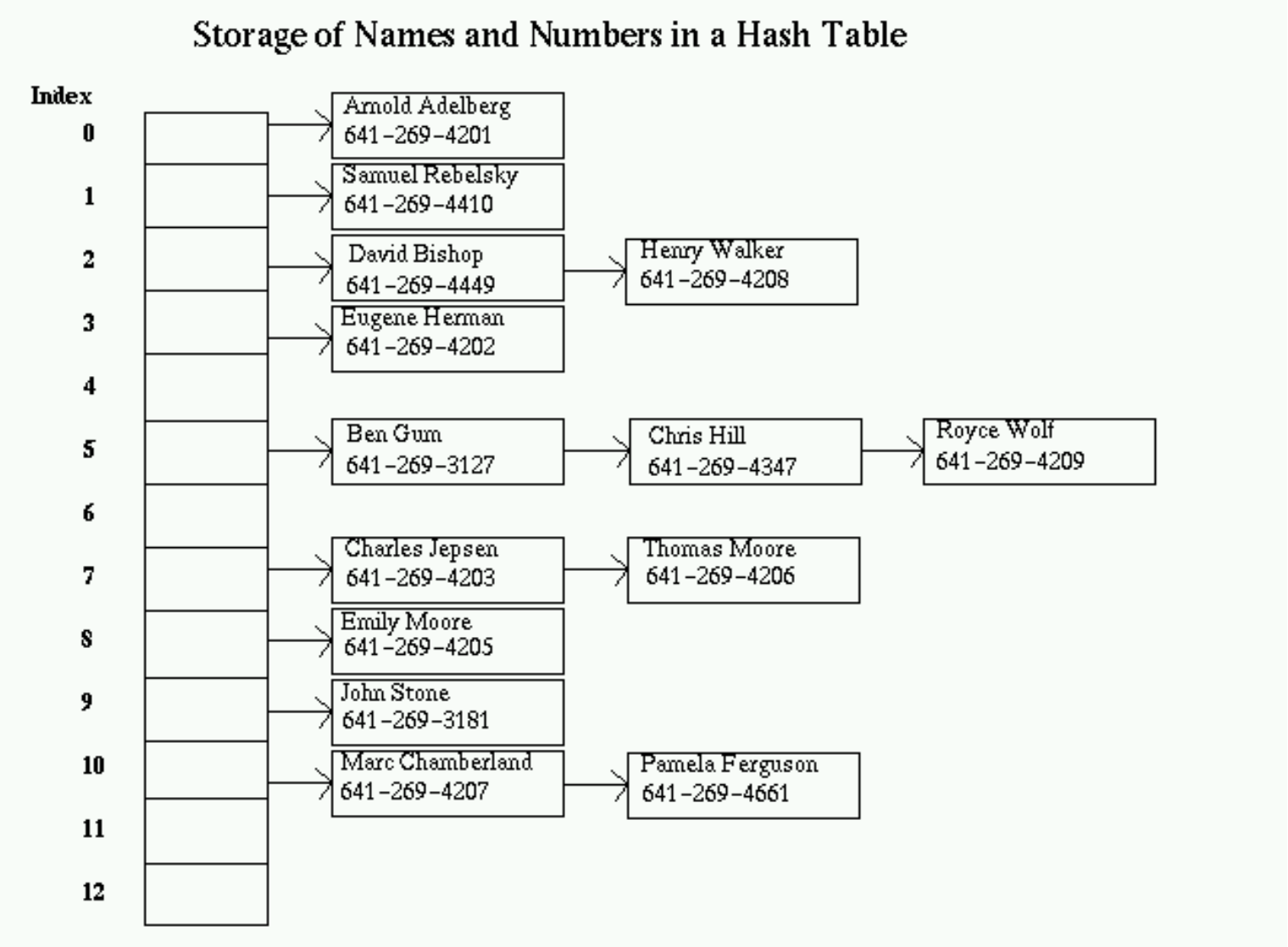
Kolizije i birthday paradox

- Verovatnoća kolizije je neintuitivan (nelinearan) fenomen, mnogo je veća nego što nam to intuicija govori
- Ako je u sobi 367 ljudi tada postoje bar dve rođene istog dana (Dirihleov princip)
- Koliko osoba mora biti u sobi da bi sa verovatnoćom 0.5 mogli reći da su dve osobe rođene istog dana?

23

- Ako je u sobi 70 ljudi tada sa verovatnoćom 0.999 postoje bar dve rođene istog dana
- 35 ljudi, verovatnoća 0.8

Otvoreno hešovanje



Veličina heš tabele: biramo neki veliki prost broj

ADT skup, otvoreno hešovanje

```
public class OHashSet<T> implements Set<T> {

    private static final int DEFAULT_TABLE_SIZE = 101;

    private static class Node {
        Object value;
        Node next;

        public Node(Object value) {
            this.value = value;
        }
    }

    private Node[] table;

    public OHashSet() {

    public OHashSet(int size) {

    private int hash(T o) {

     * Pretrazuje listu kolizija za datu vrednost hash funkcije (drugi argument metoda)
    private Node[] searchColissionChain(T element, int hashValue) {

    public boolean insert(T element) {

    public boolean remove(T element) {

    public boolean member(T element) {

    public void print() {
}
```

Čvor u listi objekata sa istom heš vrednošću

Object value – objekat

Node next – referenca na sledeći objekat u listi

Niz listi

Konstruktori

Heš funkcija

Metode propisane interfejsom Set<T>

Metoda koja štampa skup

Konstruktori i heš funkcija

```
private Node[] table;

public OHashSet() {
	this(DEFAULT_TABLE_SIZE);
}

public OHashSet(int size) {
	if (size <= 0)
		throw new IllegalArgumentException(
			"Velicina hash tabele ne moze biti negativna"
		);

	table = new Node[size];
}

private int hash(T o) {
	if (o == null)
		throw new IllegalArgumentException(
			"Hash funkcija se ne moze racunati za null objekat"
		);

	return Math.abs(o.hashCode() % table.length);
}
```

Pretraživanje lanca kolizija

```
/**
  * Pretrazuje listu kolizija za datu vrednost hash funkcije
  * (drugi argument metoda) trazeci element (prvi argument metoda).
  * Metod vraca null ukoliko element ne postoji u tabeli, ili
  * niz od dve reference:
  * 1. prva referenca je referenca na trazeni element
  * 2. druga referenca je referenca na element ispred trazenog elementa
  */
private Node[] searchColissionChain(T element, int hashValue) {
	Node current = table[hashValue];
	Node prev = null;

	while (current != null) {
		if (current.value.equals(element)) {
			Node[] ret = new Node[2];
			ret[0] = current;
			ret[1] = prev;
			return ret;
		}

		prev = current;
		current = current.next;
	}

	return null;
}
```

member i insert

```
public boolean member(T element) {
	return searchColissionChain(element, hash(element)) != null;
}

public boolean insert(T element) {
	int hashValue = hash(element);
	Node[] n = searchColissionChain(element, hashValue);
	if (n != null)
		return false;

	Node newElement = new Node(element);
	newElement.next = table[hashValue];
	table[hashValue] = newElement;

	return true;
}
```

remove

1. Proveri li da element postoji u odgovarajućem lancu kolizija
2. Ako postoji razlikujemo dva slučaja: element koji se briše je (1) na početku lanca kolizija ili (2) u sredini/na kraju lanca kolizija

```
public boolean remove(T element) {
	int hashValue = hash(element);
	Node[] n = searchColissionChain(element, hashValue);
	if (n == null)
		return false;

	if (n[0] == table[hashValue]) {
		table[hashValue] = table[hashValue].next;
	} else {
		n[1].next = n[0].next;
	}

	return true;
}
```

```
public void print() {
	for (int i = 0; i < table.length; i++) {
		System.out.print("HashCode = " + i + ": ");

		Node head = table[i];
		if (head == null) {
			System.out.println(" empty");
		} else {
			while (head != null) {
				System.out.print("[" + head.value + "]");
				head = head.next;
			}
			System.out.println();
		}
	}
}
```

Primer 1. Skup celih brojeva

```
public class ExIntegerSet {
	public static void main(String[] args) {
		OHashSet<Integer> set = new OHashSet<Integer>(5);

		for (int i = 0; i < 24; i++) {
			set.insert(i);
		}
		set.print();

		if (set.insert(10))
			S.o.p("10 ubacen u skup");
		else
			S.o.p("10 je vec u skupu");

		S.o.p(
			"Member(10) - " + set.member(10) +
			", Member(34) - " + set.member(34)
		);

		S.o.p("Remove(22) - " + set.remove(22));
		S.o.p("Member(22) - " + set.member(22));
		set.print();
	}
}
```

```
HashCode = 0: [20][15][10][5][0]
HashCode = 1: [21][16][11][6][1]
HashCode = 2: [22][17][12][7][2]
HashCode = 3: [23][18][13][8][3]
HashCode = 4: [19][14][9][4]
10 je vec u skupu
Member(10) - true, Member(34) - false
Remove(22) - true
Member(22) - false
HashCode = 0: [20][15][10][5][0]
HashCode = 1: [21][16][11][6][1]
HashCode = 2: [17][12][7][2]
HashCode = 3: [23][18][13][8][3]
HashCode = 4: [19][14][9][4]
```

Skup objekata klase *A*

- Da bi napravili skup objekata klase A koristeći *OHashSet*, klasa A treba da redefiniše sledeće metode iz klase *Object*
  - int hashCode()
  - boolean equals(Object o)
  - String toString()

- hashCode se koristi pri računanju vrednosti heš funkcije
- Metode member i remove koriste equals za poređenje elemenata
- Metoda print koristi S.O.P koji koristi toString

Primer 2. Skup studenata

```
class Student {
	private String index, fName, lName;

	public Student(String index, String fName, String lName) {
		this.index = index;
		this.fName = fName;
		this.lName = lName;
	}

	public String toString() {
		return index + ", " + fName + " " + lName;
	}

	public boolean equals(Object anotherStudent) {
		if (anotherStudent != null && anotherStudent instanceof Student)
		{
			Student s = (Student) anotherStudent;
			return index.equals(s.index) &&
				fName.equals(s.fName) &&
				lName.equals(s.lName);
		} else {
			return false;
		}
	}

	public int hashCode() {
		return toString().hashCode();
	}
}
```

... nastavak

```
public class ExStudentSet {
	public static void main(String[] args) {
		OHashSet<Student> s = new OHashSet<Student>(5);
		s.insert(new Student("264/03", "Stojan", "Markovic"));
		s.insert(new Student("145/03", "Jovana", "Lackovic"));
		s.insert(new Student("264/03", "Stojan", "Markovic"));
		s.print();
	}
}
```

```
HashCode = 0:  empty
HashCode = 1: [145/03, Jovana Lackovic][264/03, Stojan Markovic]
HashCode = 2:  empty
HashCode = 3:  empty
HashCode = 4:  empty
```

ADT skup, zatvoreno hešovanje

- Svaka ćelija u heš tabeli će imati jedno od sledeća tri stanja
  - EMPTY – prazna, slobodna ćelija
  - OCCUPIED – zauzeta ćelija, ćelija koja sadrži element
  - DELETED – ćelija koja je nekada sadržala neki element koji je u međuvremenu obrisan

- Dodavanje novog elementa *n*
  - *hash* = vrednost heš funkcije za *n*
  - Ako je *table*[*hash*] u stanju EMPTY tada
    table[*hash*] = *n, table*[*hash*] prelazi u stanje OCCUPIED
  - Inače, traži neko drugo mesto za *n*
  - **Traženje nekog drugog mesta za element se još zove probavanje (engl. *probing*)**

Strategije probavanja

- *M* = veličina heš tabele
- *hash* = vrednost heš funkcije za element *n*
- Linearno probavanje
  - *test* = (*hash* + i) % *M*, i = 1, 2, …
  - *hash* + 1, *hash* + 2, *hash* + 3, ..., 0, 1, 2, ..., hash - 1
- Linearno probavanje uzrokuje klastere u heš tabeli što usporava pretraživanje
- **Kvadratno probavanje**
  - ***test* = (*hash* + i^2) % *M*, i = 1, 2, …**
  - **hash + 1, hash + 4, hash + 9, …, hash - *k***

Lanac kolizija

- *M* = veličina heš tabele
- *hash* = vrednost heš funkcije za element *n*
- Lanac kolizija za element *n*
  - *table*[*hash*] u statusu OCCUPIED ili DELETED
  - *table*[(*hash* + 1) % *M*] u statusu OCCUPIED ili DELETED
  - *table*[(*hash* + 4) % *M*] u statusu OCCUPIED ili DELETED
  - …
  - *table*[(*hash* + i^2) % *M*] u statusu OCCUPIED ili DELETED
  - *table*[(*hash* + $(i + 1)^2$) % *M*] u statusu EMPTY

- **Lanac kolizija je uslovljen strategijom probavanja, može da sadrži elemente koji imaju različitu vrednost heš funkcije**

Operacije

- Pretraživanje heš tabele
  - pretraživanje lanca kolizija za dati element
- Dodavanje novog elementa u heš tabelu
  - dodavanje na kraj lanca kolizija, ili
  - dodavanje u prvu ćeliju u statusu DELETED u lancu kolizija
- Brisanje elementa iz heš tabele
  - Lenjo brisanje – ćelija tabele koja je sadržala element promeni status u DELETED

Lanac kolizija

- Heš tabela je niz neke dužine (*M*, neki veliki **prost broj**)
  → lanac kolizija ima maksimalnu dužinu

- **Za maksimalnu dužinu lanca kolizicija kod kvadratnog probavanja ćemo uzeti (*M* – 1) / 2**

- Prvih (M - 1) / 2 kvadratnih probavanja, gde je M prost broj, su uvek probavanja različitih lokacija u tabeli

- Posle (M – 1) / 2 neuspešnih probavanja proširujemo tabelu

- Proširenje tabele takođe radimo kada je **opterećenje** heš tabele veće od 70%
  - više od 70% ćelija su u stanju OCCUPIED

Proširivanje tabele

- Veličina tabele: najmanji prost broj koji je barem dva puta veći nego što je trenutna veličina tabele

1. Napravimo kopiju stare tabele i statusa
2. Realociramo tabelu i postavimo statuse svih ćelija ne EMPTY
3. Iteriramo kroz staru tabelu i ćelije u statusu OCCUPIED dodajemo u novu tabelu
4. Staru tabelu i statuse postavimo na *null* vrednosti

Klasa CHashSet

```
public class CHashSet<T> implements Set<T> {

	private static final int DEFAULT_TABLE_SIZE = 101;

	private enum Status {
		EMPTY,
		OCCUPIED,
		DELETED
	};

	private Object[] table;		// tabela
	private Status[] status;	// statusi celija
	private int numElements;	// broj elemenata (OCCUPIED celija) u tabeli
```

- Nizovi status i table će imati istu dužinu
- status[i] predstavlja status ćelije table[i]

```
public CHashSet() {
	this(DEFAULT_TABLE_SIZE);
}

public CHashSet(int size) {
	if (size <= 0) {
		throw new IllegalArgumentException(
			"Velicina hash tabele ne moze biti negativna"
		);
	}

	reset(size);
}

private void reset(int size) {
	table = new Object[size];
	status = new Status[size];
	for (int i = 0; i < status.length; i++) {
		status[i] = Status.EMPTY;
	}
	numElements = 0;
}

private int hash(T o) {
	if (o == null)
		throw new IllegalArgumentException(
			"Hash funkcija se ne moze racunati za null objekat"
		);

	return Math.abs(o.hashCode() % table.length);
}
```

Pretraživanje lanca kolizija

```
/**
 * Pretrazuje lanac kolizija.
 * Vraca poziciju elementa ukoliko postoji u lancu kolizija,
 * inace vraca -1
 */
private int searchCollisionChain(T element, int hashValue) {
	int currentPosition = hashValue;
	int i = 0;
	int maxChainLength = (table.length - 1) / 2;

	while (i <= maxChainLength && status[currentPosition] != Status.EMPTY) {
		if (status[currentPosition] == Status.OCCUPIED &&
			table[currentPosition].equals(element))
		{
			return currentPosition;
		}

		i++;
		currentPosition = (hashValue + i * i) % table.length;
	}

	return -1;
}
```

member i remove

- Obe metode se oslanjaju na prethodno opisanu metodu za pretraživanje lanca kolizija

```
public boolean member(T element) {
	return searchCollisionChain(element, hash(element)) >= 0;
}

public boolean remove(T element) {
	int pos = searchCollisionChain(element, hash(element));
	if (pos < 0)
		return false;

	status[pos] = Status.DELETED;
	numElements--;

	return true;
}
```

```java
public boolean insert(T element) {
	int hashValue = hash(element);

	int maxChainLength = (table.length - 1) / 2;
	int i = 0;
	boolean endOfChain = false;
	int firstAvailablePosition = -1;

	while (!endOfChain && i <= maxChainLength) {
		int currentPosition = (hashValue + i * i) % table.length;

		if (status[currentPosition] == Status.OCCUPIED) {
			if (table[currentPosition].equals(element)) {
				return false;
			}
		}
		else {
			if (firstAvailablePosition == -1) {
				firstAvailablePosition = currentPosition;
			}

			if (status[currentPosition] == Status.EMPTY) {
				endOfChain = true;
			}
		}

		i++;
	}
	// to be continued...
```

```
        ...
        if (firstAvailablePosition == -1 || loadFactor() > 0.7) {
                expand();
                hashValue = hash(element);
                add(element, hashValue);
        } else {
                status[firstAvailablePosition] = Status.OCCUPIED;
                table[firstAvailablePosition] = element;
                numElements++;
        }

        return true;
}

private void add(T element, int hashValue) {
        boolean foundPosition = false;
        int i = 0;

        while (!foundPosition) {
                int currentPosition = (hashValue + i * i) % table.length;

                if (status[currentPosition] != Status.OCCUPIED) {
                        status[currentPosition] = Status.OCCUPIED;
                        table[currentPosition] = element;
                        numElements++;
                        foundPosition = true;
                } else {
                        i++;
                }
        }
}
```

Proširenje tabele

```
private void expand() {
	int oldSize = table.length;
	int size = 2 * oldSize;
	while (!isPrime(size))
		size++;

	System.out.println("Expanding to size - " + size);

	Object[] oldTable = new Object[oldSize];
	Status[] oldStatus = new Status[oldSize];

	for (int i = 0; i < oldSize; i++) {
		oldTable[i] = table[i];
		oldStatus[i] = status[i];
	}

	reset(size);

	for (int i = 0; i < oldSize; i++) {
		if (oldStatus[i] == Status.OCCUPIED) {
			T elem = (T) oldTable[i];
			add(elem, hash(elem));
		}
	}

	oldTable = null;
	oldStatus = null;
}
```

Ostatak metoda klase CHashSet

```
private boolean isPrime(int num) {
	for (int i = 2; i * i <= num; i++) {
		if (num % i == 0)
			return false;
	}

	return true;
}

private double loadFactor() {
	return (double) numElements / (double) table.length;
}

public void print() {
	for (int i = 0; i < table.length; i++) {
		if (status[i] == Status.EMPTY) {
			System.out.print("[E]");
		} else if (status[i] == Status.DELETED) {
			System.out.print("[D]");
		} else {
			System.out.print("[O - " + table[i] +"]");
		}
	}
	System.out.println();
}
```

Primer lanca kolizija

```
public class ExIntegerSetCH {
	public static void main(String[] args) {
		CHashSet<Integer> set = new CHashSet<Integer>(11);

		// formiramo brojeve koji imaju istu hash vrednost - 2
		for (int i = 0; i < 5; i++) {
			int num = 2 + 11 * i;
			System.out.println("Dodajem: " + num);
			set.insert(num);
			set.print();
		}
	}
}
```

```
Dodajem: 2
[E][E][O - 2][E][E][E][E][E][E][E][E]
Dodajem: 13
[E][E][O - 2][O - 13][E][E][E][E][E][E][E]
Dodajem: 24
[E][E][O - 2][O - 13][E][E][O - 24][E][E][E][E]
Dodajem: 35
[O - 35][E][O - 2][O - 13][E][E][O - 24][E][E][E][E]
Dodajem: 46
[O - 35][E][O - 2][O - 13][E][E][O - 24][O - 46][E][E][E]
```

ADT mapa, otvoreno hešovanje

- Čvorovi u tabeli sada sadrže parove (ključ, vrednost)
- Heš funkcija se primenjuje samo na ključeve
  - iliti, lanac kolizija sadrži one elemente čiji ključevi imaju istu vrednost heš funkcije

```
public class OHashMap<K, V> implements Map<K, V> {

	private static final int DEFAULT_TABLE_SIZE = 101;

	private static class Node {
		Object key;
		Object value;
		Node next;

		public Node(Object key, Object value) {
			this.key = key;
			this.value = value;
		}
	}

	private Node[] table;
	//... to be continued
```

Konstruktori i heš funkcija

```
public OHashMap() {
	this(DEFAULT_TABLE_SIZE);
}

public OHashMap(int size) {
	if (size <= 0)
		throw new IllegalArgumentException(
			"Velicina hash mape ne moze biti negativna"
		);

	table = new Node[size];
}

private int hash(K o) {
	if (o == null)
		throw new IllegalArgumentException(
			"Hash funkcija se ne moze racunati za null objekat"
		);

	return Math.abs(o.hashCode() % table.length);
}
```

Pretraživanje lanca kolizija

```
/**
 * Pretrazuje listu kolizija trazeci element odredjen kljucem.
 * Ukoliko element postoji u tabeli tada se vraca niz od dve reference
 * - prva je pokazivac na element u listi kolizija
 * - druga je pokazivac na prethodni element
 */
private Node[] searchCollisionChain(K key, int hashValue) {
	Node current = table[hashValue];
	Node prev = null;
	while (current != null) {
		if (current.key.equals(key)) {
			Node[] ret = new Node[2];
			ret[0] = current;
			ret[1] = prev;
			return ret;
		}

		prev = current;
		current = current.next;
	}

	return null;
}
```

get

- U praksi se često dešava da postoje elementi koji se dobavljaju mnogo češće u odnosu na prosek (***hot data*** **i** ***cold data***)
- LRU (*least recently used*) strategija
  - poslednji traženi element pomeri na početak lanca kolizija

```
public V get(K key) {
        int hashValue = hash(key);
        Node[] n = searchCollisionChain(key, hashValue);
        if (n == null)
                return null;

        // LRU strategija - stavi element na pocetak lanca kolizija
        if (n[0] != table[hashValue]) {
                n[1].next = n[0].next;
                n[0].next = table[hashValue];
                table[hashValue] = n[0];
        }

        return (V) n[0].value;
}
```

```
public boolean insert(K key, V value) {
	int hashValue = hash(key);

	if (searchCollisionChain(key, hashValue) != null) {
		return false;
	}

	Node newElement = new Node(key, value);
	newElement.next = table[hashValue];
	table[hashValue] = newElement;

	return true;
}

public boolean delete(K key) {
	int hashValue = hash(key);

	Node[] n = searchCollisionChain(key, hashValue);
	if (n == null)
		return false;

	if (n[0] == table[hashValue]) {
		table[hashValue] = table[hashValue].next;
	} else {
		n[1].next = n[0].next;
	}

	return true;
}
```

```
public boolean modify(K key, V value) {
	int hashValue = hash(key);
	Node[] n = searchCollisionChain(key, hashValue);
	if (n[0] != null) {
		n[0].value = value;
		return true;
	} else {
		return false;
	}
}

public void print() {
	for (int i = 0; i < table.length; i++) {
		S.o.p("HashCode = " + i + ": ");

		Node head = table[i];
		if (head == null) {
			S.o.p(" empty");
		} else {
			while (head != null) {
				S.o.p("[" + head.key + ": " + head.value + "]");
				head = head.next;
			}
			System.out.println();
		}
	}
}
```

Primer...

```
class Person {
	private String name;
	private int age;

	public Person(String name, int age) {
		this.name = name;
		this.age = age;
	}

	public String toString() {
		return name + ", " + age;
	}
}

public class ExPersonMap {
	public static void main(String[] args) {
		OHashMap<String, Person> map = new OHashMap<String, Person>(5);
		map.insert("1235", new Person("Pera", 24));
		map.insert("2896", new Person("Mika", 21));
		map.insert("3521", new Person("Zika", 23));
		map.insert("3521", new Person("Fica", 26));
		map.insert("2225", new Person("Mica", 22));
		map.insert("9862", new Person("Tika", 28));
		map.print();
		System.out.println("Osoba 1235: " + map.get("1235"));
		map.print();
	}
}
```

```java
public class ExPersonMap {
	public static void main(String[] args) {
		OHashMap<String, Person> map = new OHashMap<String, Person>(5);
		map.insert("1235", new Person("Pera", 24));
		map.insert("2896", new Person("Mika", 21));
		map.insert("3521", new Person("Zika", 23));
		map.insert("3521", new Person("Fica", 26));
		map.insert("2225", new Person("Mica", 22));
		map.insert("9862", new Person("Tika", 28));
		map.print();
		System.out.println("Osoba 1235: " + map.get("1235"));
		map.print();
	}
}
```

```
HashCode = 0:  empty
HashCode = 1:  empty
HashCode = 2: [9862: Tika, 28][2896: Mika, 21]
HashCode = 3: [2225: Mica, 22][3521: Zika, 23][1235: Pera, 24]
HashCode = 4:  empty
Osoba 1235: Pera, 24
HashCode = 0:  empty
HashCode = 1:  empty
HashCode = 2: [9862: Tika, 28][2896: Mika, 21]
HashCode = 3: [1235: Pera, 24][2225: Mica, 22][3521: Zika, 23]
HashCode = 4:  empty
```