Програмування-1

Лекція 12

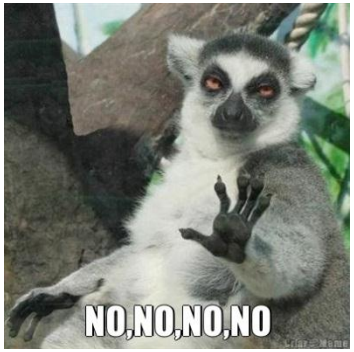
Колекції: HashSet, Map

Інтерфейс **Set**

- Не допускає додавання дублікатів
- Слід використовувати виключно для immutable об’єктів!
- Є нащадком Collection
  - усі методи такі ж самі як у Collection
    - додатково є контракт на унікальність елементів
- Порядок елементів при перегляді в загальному випадку не гарантується
    - **SortedSet** (інтерфейс) – елементи відсортовані
    - **LinkedHashSet** (класс) – елементи у тому порядку в якому додавались
- Реализації: **TreeSet**, **HashSet**, **LinkedHashSet**, …

Як гарантувати відсутність дублікатів у колекції?

- Відповідь:
  - перед додаванням провести пошук
  - не додавати, якщо такий елемент вже є



- Пошук має бути швидким!!!

Алгоритми пошуку

	Лінійний пошук	Двійковий пошук	Хешування
Швидкість (найгірший випадок)	O(n)	O($\log_2 n$)	Залежіть від налаштувань Як правило швидший за двійковий
Вимоги до незмінності об’єктів	немає	immutable	immutable
Вимоги до функціональності	equals()	Comparable or Comparator	equals() + hashCode()

Клас **HashSet**

- Дублікати не допускаються (implements Set)
- Для пошуку та додавання використовується **хешування** (не плутати з кешуванням)
- Порядок елементів при перегляді ітератором – непередбачуваний
- Допускається запис значення **null**
- Допускається додавання об’єктів різних типів
- У об’єктів має бути правильні методи
  - **equals**(...) та **hashCode**()
- Коректно працює лише з immutable об’єктами
- Українська: хеш = геш 

Клас HashSet - приклад

```
import java.util.*;

public class HashSetDemo{

        public static void main(String[] args) {

                Set set = new HashSet(); // try LinkedHashSet

                set.add("one");
                set.add("two");
                set.add("three");
                set.add("four");
                set.add("one");
                set.add("two");
                // try this:
                //set.add(new Integer(1)); // No compare = No Exception
                //set.add(null); // No NullPointerException

                // check order and duplicates
                System.out.println(set);
        }

}
```

Хешування

- Хешування – спосіб прискорення пошуку

- Приклад хешування – телефонний записник:
  - при записі – пишемо на потрібну сторінку
  - при пошуку – шукаємо ЛИШЕ на потрібній сторінці
  - якщо є на сторінці – знайшли швидко
  - якщо на потрібній сторінці немає – пошук завершено

- Хеш-функція – алгоритм, за яким на основі стану об’єкту обчислюється його хеш-код

- Хеш-код – значення, яке визначає на яку «сторінку» всередині колекції потрібно записати об’єкт

Вимоги до equals та hashCode

- **equals**

Рефлексивність	x.equals(x) == true
Симетричність	Якщо x.equals(y) == true, то y.equals(x) == true
Транзитивність	Якщо x.equals(y) == true, та y.equals(z) == true, то x.equals(z) == true
Консистентність	Послідовні виклики x.equals(y) мають весь час повертати або true або false, якщо об'єкти x та y не змінювались
Порівняння з null	x.equals(null) == false

- **hashCode**

ОБОВ'ЯЗКОВО!	Послідовні виклики x.hashCode() мають повертати одне й те саме значення, якщо об'єкт x не змінювався (це значення може змінюватись при рестарті програми)
ОБОВ'ЯЗКОВО!	Якщо x.equals(y) == **true**, то x.hashCode() **==** y.hashCode()
БАЖАНО	Якщо x.equals(y) == **false**, то x.hashCode() **!=** y.hashCode()

Приклади хеш-функцій

```
class Point {
        int x; // x = 0..100
        int y; // y = 0..100
        ...
        public int hashCode() {
                return ...
        }
}
```

Гарна хеш-функція:
- швидко обчислюється
- має низький % колізій

Невірна хеш-функція	`(int)(Math.random()*100);`
Правильна, але погана	`42; // усі об'єкти в одно відро`
Не дуже гарна	`x+y; // у деяких об'єктів буде колізія`
Гарна	`x * 97 + y; // у різних об'єктів - різні`

Приклад

Є клас:

```
class Person {
        String name;
        String surname;
}
```

Зробити так, щоб об’єкти цього класу можна було зберігати у HashSet

Ще кілька термінів

English	Українська	Приклад
Key, K	ключ	прізвище у записнику
Value, V	значення	телефон у записнику
bucket	відро	сторінка у записнику
hash, hash code	хеш, хеш-код	перша літера прізвища
hash function (hashCode())	хеш-функція	беремо першу літеру прізвища
capacity	ємність	кількість сторінок у записнику
load factor >= size/capacity	максимальне значення кількість / ємність	якщо висока щільність записів на сторінках, то потрібен новий записник
collision	колізія	два та більше прізвищ на одній сторінці

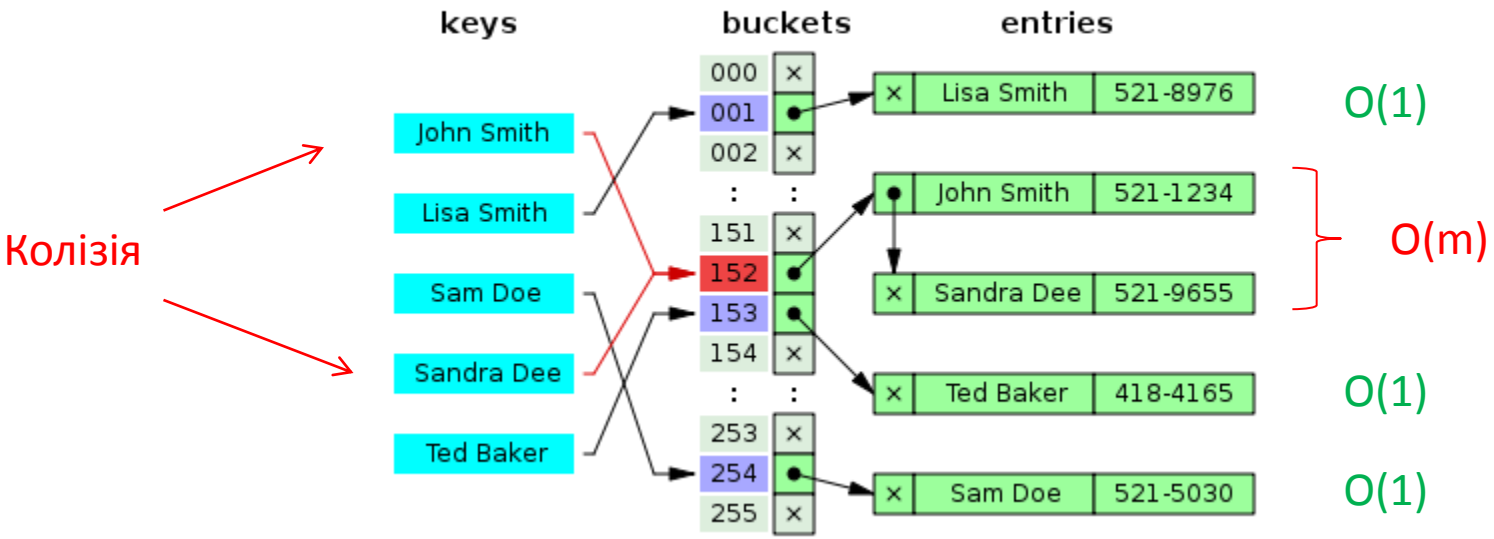
Capacity, Load Factor

Конструктор	Initial Capacity	Load Factor
HashSet()	16	0,75
HashSet(Collection c)	> c.size()	0,75
HashSet(int initialCapacity)	initialCapacity	0,75
HashSet(int initialCapacity, float loadFactor)	initialCapacity	loadFactor

Реалізація HashSet

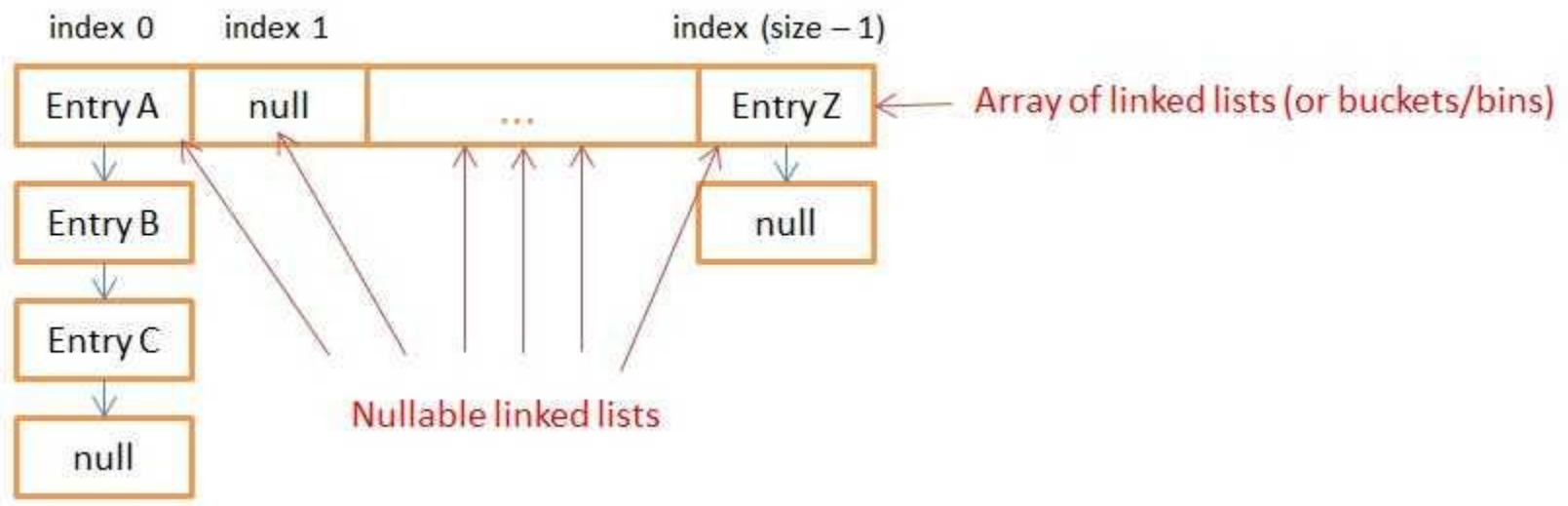
- HashSet реалізований через HashMap ☺
  - btw, TreeSet реалізували через TreeMap ☺☺
- Алгоритм додавання/пошуку об’єкта:
  - По об’єкту розраховуємо хеш
  - По хешу знаходимо потрібне відро
  - У відрі - посилання на зв’язаний список
  - Шукаємо/додаємо об’єкт у зв’язаний список
  - Якщо size/capacity > load factor, то resize

Реалізація HashSet (JDK7-: Array + Linked Lists)



Колізій немає: O(1)
Колізії є: O(m)

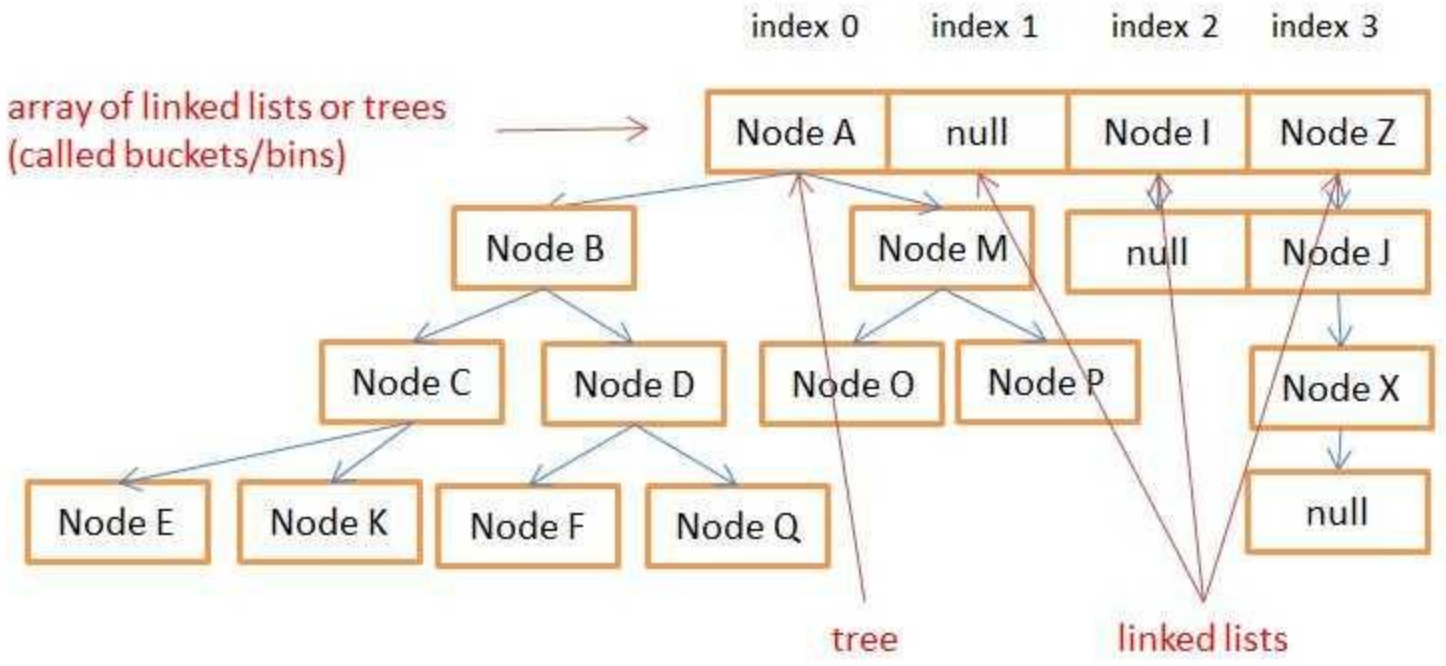
Реалізація HashSet (JDK7-: Array + Linked Lists)



Колізій немає: O(1)
Колізії є: O(m)

Реалізація HashSet
(JDK8+: Array + Tree / Linked Lists)

Реалізація HashSet (JDK8+: Array + Tree / Linked Lists)



Колізій немає: O(1)
Колізії є, елементи Comparable: $O(\log_2 m)$
Колізії є, елементи Not Comparable: O(m)

HashSet vs TreeSet

	HashSet	TreeSet
Спосіб порівняння елементів	hashCode() + equals()	Comparable.compareTo() Comparator.compare()
Сортування при перегляді	-	+
.add(**null**)	+	Comparable: - Comparator: +/-
Швидкодія	++ O(1)	+ $O(\log_2 n)$
Внутрішня реалізація	HashMap	TreeMap

Map

- Map – асоціативний масив
- Map<K,V>
  - K – Key
  - V – Value
- Map<K,V> це інтерфейс
  - TreeMap<K,V> - клас
  - HashMap<K,V> - клас
- Entry<K,V> - пара «ключ-значення»

Map<K,V>

V	put(K key, V value)
V	get(Object key)
V	remove(Object key)
boolean	containsKey(Object key)
boolean	containsValue(Object value)

void	clear()
boolean	isEmpty()
int	size()

Set<K>	keySet()
Collection<V>	values()
Set<Map.Entry<K,V>>	entrySet()

void	putAll(Map<? extends K,? extends V> m)

boolean	equals(Object o)
int	hashCode()

Map.Entry<K,V>

K	getKey()
V	getValue()
V	setValue(V value)

boolean	equals(Object o)
int	hashCode()

Map - приклад

```
HashMap<String, String> map = new HashMap<>();

map.put("Ivanov", "555-1234");
map.put("Petrov", "555-0000");
map.put("Ivanov", "555-4242");
map.put("Petrova", "555-0000");

System.out.println(map.size());
System.out.println(map.get("Ivanov"));

for (String surname : map.keySet()) {
   System.out.println(surname);
}

for (String phone : map.values()) {
   System.out.println(phone);
}

for (Map.Entry<String, String> entry : map.entrySet()) {
   System.out.println(entry.getKey() + " : " + entry.getValue());
}
```

```
3

555-4242

Petrov
Petrova
Ivanov

555-0000
555-0000
555-4242

Petrov : 555-0000
Petrova : 555-0000
Ivanov : 555-4242
```

SubList, SubSet

- Створення підколекції з колекції
- Зміни у підколекції змінюють основну колекцію

```
List list = new ArrayList();
list.add("a");
list.add("b");
list.add("c");
list.add("d");
list.add("e");
list.add("f");

System.out.println("list: " + list);
List view = list.subList(2, 5);

System.out.println("view: " + view);
view.remove(1);
System.out.println("view: " + view);
System.out.println("list: " + list);
```

```
output:
list: [a, b, c, d, e, f]

view: [c, d, e]

view: [c, e]
list: [a, b, c, e, f]
```

Колекції в JDK 1.0

- Основні представники:
  - **Vector** – масив змінної довжини. Використовуйте **ArrayList**
  - **Stack** – стек FILO. Використовуйте **Deque**
  - **Hashtable** – асоціативний масив. Використовуйте **Map**
    - було заборонено використовувати null як ключ
  - **Enumeration** – перегляд колекції. Використовуйте **Iterator**
    - не дозволялось видаляти елементи при перегляді

- Старі колекції - thread-safe (надійні при паралельних обчисленнях але повільні)
  - Це не завжди потрібно
  - Трохи знижає швидкість
- Нові колекції за-замовчуванням не мають такої надлишковості
  - Працюють трохи швидше
- Якщо в нових колекціях потрібна надійність thread-safe – за допомогою утилітного класу Collections можна отримати thread-safe