

Коллекции TreeMap и TreeSet



Юрий
Пеньков




Юрий Пеньков

Java Software Engineer в InnoStage



План занятия

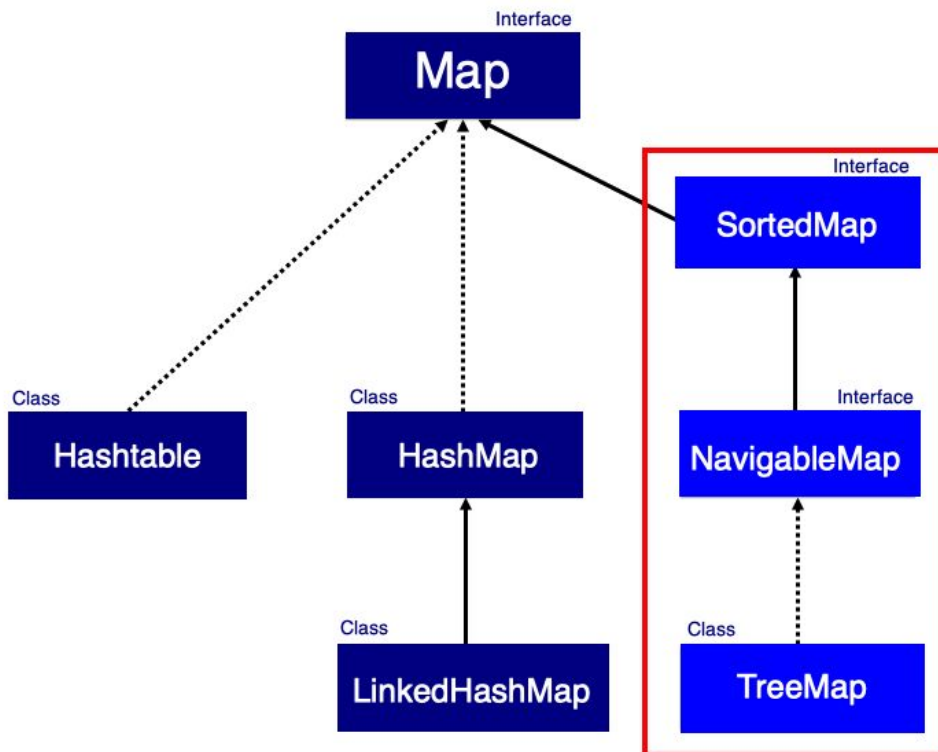
1. [Структура данных TreeMap](#)
2. [Структура данных TreeSet](#)
3. [Итоги](#)
4. [Домашнее задание](#)



Структура данных TreeMap

Структура данных TreeMap

Ранее мы рассмотрели одну из ключевых реализаций интерфейса Map – HashMap. Сейчас поговорим о еще одной реализации этого интерфейса – коллекции **TreeMap**.





Структура данных TreeMap

TreeMap — структура данных, которая хранит элементы в отсортированном по ключам виде.

Это значит, что коллекция хранит пары элементов ключ-значение. Добавлять / удалять / искать / изменять элементы можно только с использованием их ключа, по которому и происходит сортировка элементов внутри коллекции.



Устройство TreeMap

TreeMap основан на **красно-черном** дереве, вследствие чего TreeMap сортирует элементы по ключу в естественном порядке или на основе заданного при создании коллекции компаратора.

Главное, что красно-черное дерево является бинарным и в то же время сбалансированным — это гарантирует нам скорость работы структуры данных.

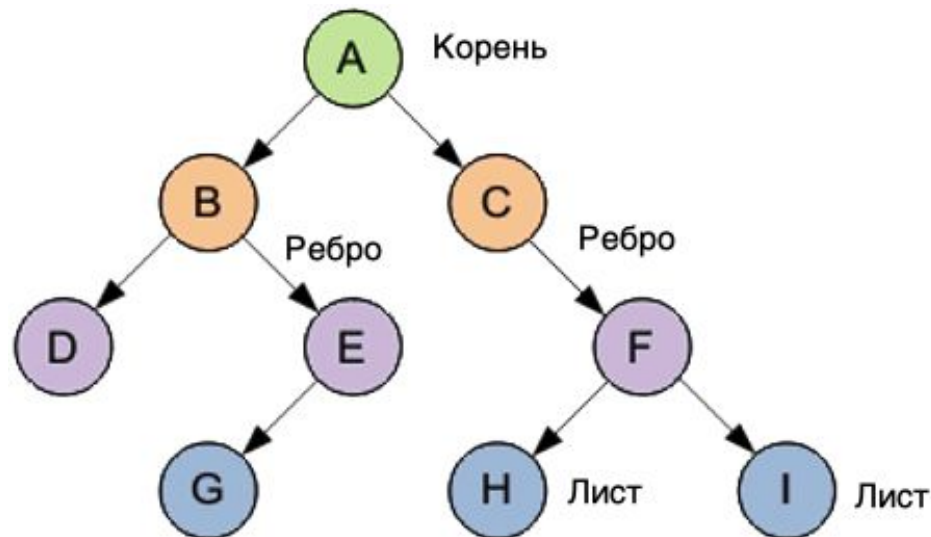


Что такое
деревья?

Ответ

Дерево — это структура данных, состоящая из **узлов** и **ребер**.

- Каждый узел соединен с другим узлом с помощью ребра.
- Существует хотя бы один узел, у которого нет ни одного входящего ребра — он называется **корень**.
- Узлы, у которых нет исходящих ребер, называются **листами**.





Красно-черное дерево

Красно-черное дерево является **бинарным** деревом.

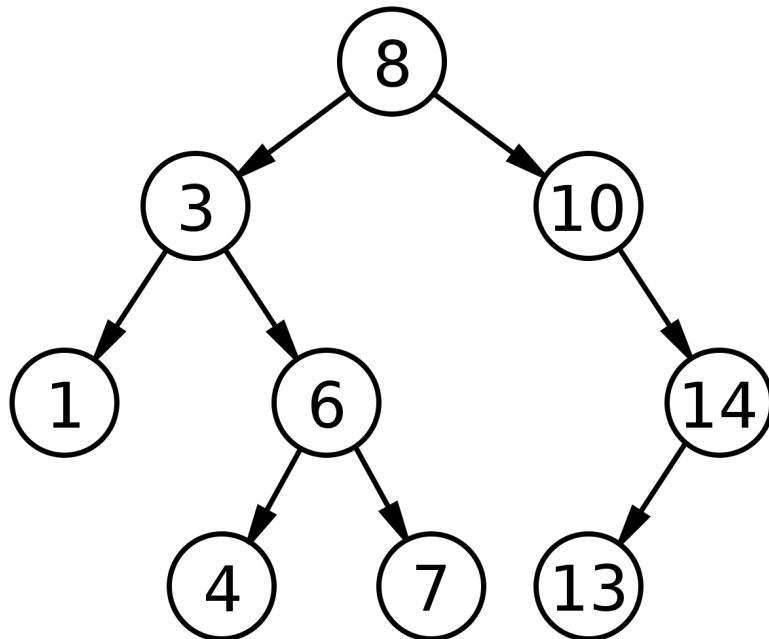


Что такое
бинарное дерево?

Ответ

Бинарное дерево (или двоичное) — это структура вида дерева, обладающая следующими свойствами:

- каждый узел может иметь не больше двух дочерних узлов;
- потомки слева всегда меньше этого узла;
- потомки справа всегда больше этого узла;

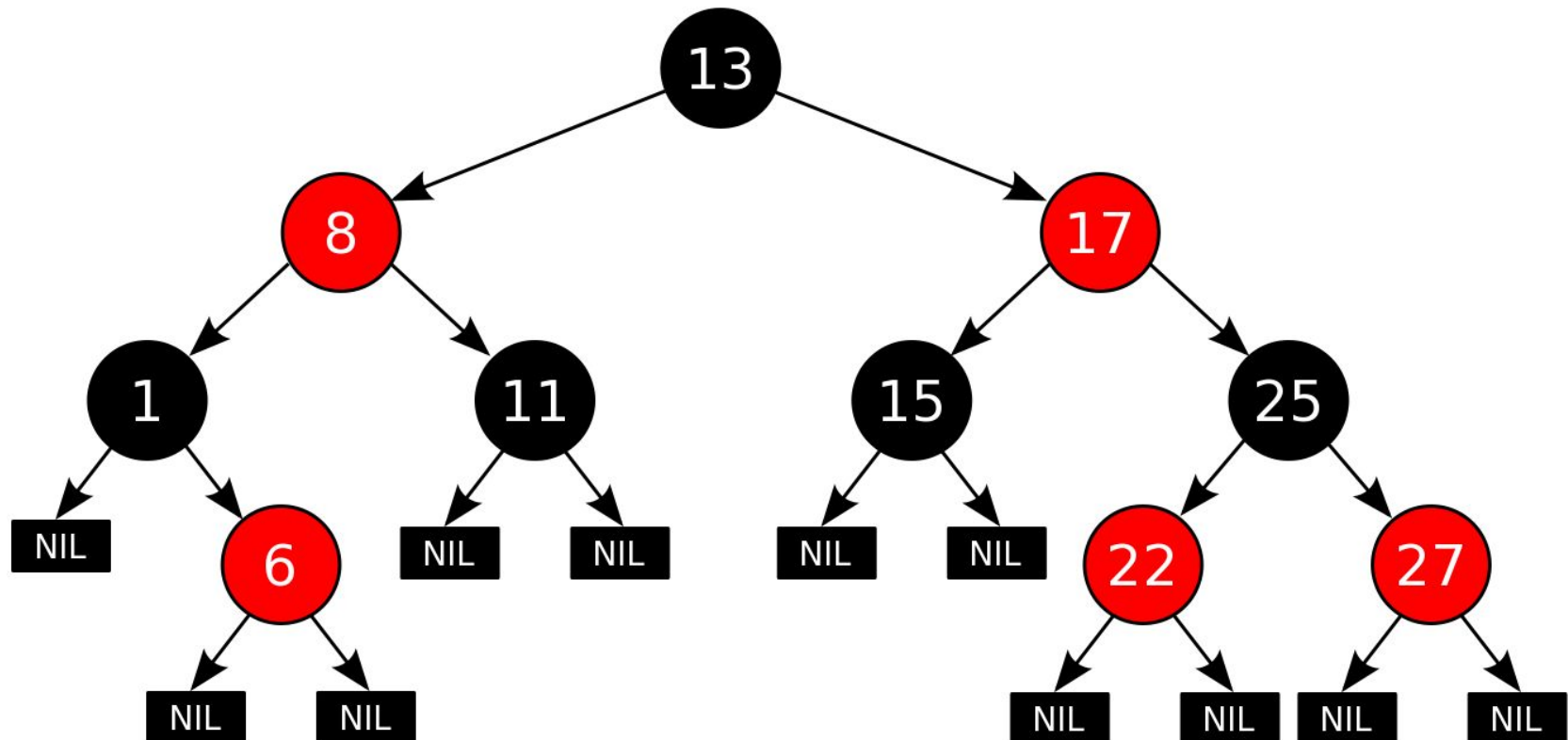


Ответ

Красно-черное дерево — это бинарное дерево, узлы которого окрашены либо в красные, либо в черные цвета.

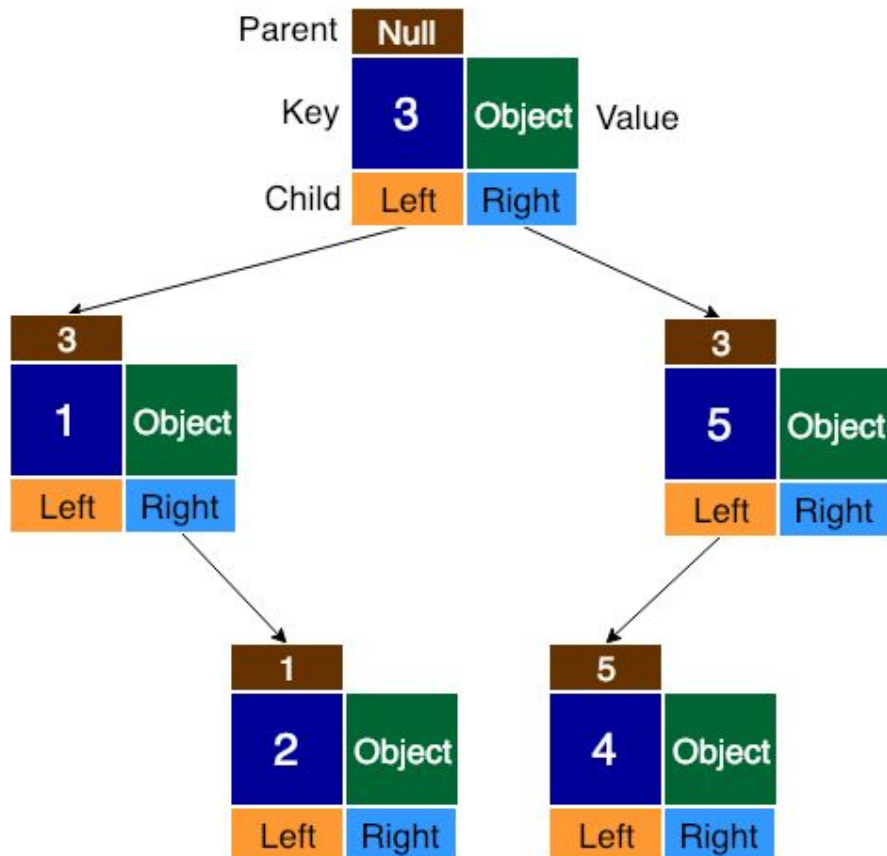
- Чтобы найти нужный элемент в дереве, нужно сравнить искомое значение с корневым элементом (элемент, у которого нет предков).
- Если искомое значение *больше* корневого элемента, то мы переходим к потомку элемента справа;
- Если искомое значение *меньше*, то переходим к потомку слева и сравниваем искомый элемент с потомком.
- Сравнение и переход происходит до тех пор, пока не будет найден нужный элемент или мы не дойдем до конца дерева (элемента лист).

Красно-черное дерево



Поиск элемента в бинарном дереве

Пример хранения объектов с ключами, значения которых находятся в интервале **от 1 до 5**.





Методы коллекции `TreeMap`

Класс `TreeMap` реализует интерфейс `NavigableMap`, который наследует методы интерфейса `SortedMap`.

Методы интерфейса SortedMap

- **K firstKey(), K lastKey()** — возвращает ключ первого или последнего элемента `map` соответственно.
- **SortedMap<K, V> headMap(K end)** — возвращает `map` до указанного ключа (включая его в выборку).
- **SortedMap<K, V> tailMap(K start)** — возвращает `map`, начиная от заданного ключа до завершения текущей коллекции.
- **SortedMap<K, V> subMap(K start, K end)** — возвращает `map` в интервале заданных ключей.

Методы интерфейса NavigableMap

- **Map.Entry<K, V> ceilingEntry(K key)** — возвращает элемент, который больше или равен ключу `key`. Если такого ключа нет, то возвращается `null`.
- **Map.Entry<K, V> floorEntry(K key)** — возвращает элемент, который меньше или равен ключу `key`. Если такого ключа нет, то возвращается `null`.
- **Map.Entry<K, V> higherEntry()** — возвращает элемент, который строго больше ключа `key`. Если такого ключа нет, то возвращается `null`.
- **Map.Entry<K, V> lowerEntry()** — возвращает элемент, который строго меньше ключа `key`. Если такого ключа нет, то возвращается `null`.
- **Map.Entry<K, V> firstEntry()** — возвращает первый элемент `map`.
- **Map.Entry<K, V> lastEntry()** — возвращает последний элемент `map`.

Методы интерфейса `NavigableMap`

- **`Map.Entry<K, V> pollFirstEntry()`** — возвращает и удаляет первый элемент из `map`.
- **`Map.Entry<K, V> pollLastEntry()`** — возвращает и удаляет последний элемент из `map`.
- **`K ceilingKey(K key)`** — возвращает ключ, который больше или равен ключу `key`. Если такого ключа нет, то возвращается `null`.
- **`K floorKey(K key)`** — возвращает ключ, который меньше или равен ключу `key`. Если такого ключа нет, то возвращается `null`.
- **`K lowerKey(K key)`** — возвращает ключ, который меньше ключа `key`. Если такого ключа нет, то возвращается `null`.
- **`K higherKey(K key)`** — возвращает ключ, который больше ключа `key`. Если такого ключа нет, то возвращается `null`.

Методы интерфейса `NavigableMap`

- **`NavigableSet descendingKeySet()`** — возвращает объект `NavigableSet`, в котором все ключи расположены в обратном порядке исходного объекта `map`.
- **`NavigableMap<K, V> descendingMap()`** — возвращает отображение `NavigableMap`, которое содержит все элементы в обратном порядке исходного объекта `map`.
- **`NavigableSet navigableKeySet()`** — возвращает объект `NavigableSet`, который содержит все ключи `map`.
- **`NavigableMap<K, V> headMap(K upperBound, boolean incl)`** — возвращает отображение `NavigableMap`, которое содержит все элементы оригинального `NavigableMap` вплоть от элемента с ключом `upperBound`. Параметр `incl` при значении `true` указывает, что элемент с ключом `upperBound` также включается в выходной набор.

Методы интерфейса `NavigableMap`

- **`NavigableMap<K, V> tailMap(K lowerBound, boolean incl)`** — возвращает отображение `NavigableMap`, которое содержит все элементы оригинального `NavigableMap`, начиная с элемента с ключом `lowerBound`. Параметр `incl` при значении `true` указывает, что элемент с ключом `lowerBound` также включается в выходной набор.
- **`NavigableMap<K, V> subMap(K lowerBound, boolean lowIncl, K upperBound, boolean highIncl)`** — возвращает отображение `NavigableMap`, которое содержит все элементы оригинального `NavigableMap` от элемента с ключом `lowerBound` до элемента с ключом `upperBound`. Параметры `lowIncl` и `highIncl` при значении `true` включают в выходной набор элементы с ключами `lowerBound` и `upperBound` соответственно.

Конструкторы коллекции **TreeMap**

- **TreeMap()** — создает пустую коллекцию `map`.
- **TreeMap(Map<K, ? extends V> map)** — создание коллекции на основе уже существующей коллекции `map`.
- **TreeMap(SortedMap<K, ? extends V> map)** — способ аналогичен предыдущему, создание коллекции на основе существующей.
- **TreeMap(Comparator<? super K> comparator)** — создание коллекции с отдельным объектом `Comparator` (необходим для сортировки элементов).

Пример использования TreeMap

Создадим коллекцию пользователей, которых будем хранить в отсортированном виде по полю **id**

Класс “Пользователь”:

```
class User {  
  
    public User(int id, String name, String surname) {  
        this.id = id;  
        this.name = name;  
        this.surname = surname;  
    }  
  
    int id;  
    String name;  
    String surname;  
  
    @Override  
    public String toString() {  
        return "User{" + id + ",name=" + name +  
            ",surname=" + surname + "}";  
    }  
}
```

Сравнение объектов в TreeMap

Порядок сортировки в `TreeMap` для строк и чисел задается естественным порядком:

String

- 1. abc
- 2. amc
- 3. bcd
- ...
- n. zzz

Числа

- 1. 12345
- 2. 19999
- 3. 23434
- ...
- n. 99999

Это значит, что для наших объектов, созданных из класса **User**, нет необходимости задавать алгоритм сортировки, так как ключ сортировки **id** — это число.

Создадим коллекцию TreeMap

```
public static void main(String[] args) {  
  
    // Создание коллекции  
    TreeMap<Integer, User> users = new TreeMap<>();  
  
    // Создание пользователей  
    User user1 = new User(1, "Иван", "Крайнов");  
    User user2 = new User(2, "Семен", "Петров");  
    User user3 = new User(3, "Алексей", "Давыдов");  
  
    // Добавление пользователей в TreeMap  
    users.put(user3.id, user3);  
    users.put(user1.id, user1);  
    users.put(user2.id, user2);  
  
    // Вывод пользователей  
    System.out.println(users);  
}
```

Результат работы программы:

```
{1=User{1,name=Иван,surname=Крайнов}, 2=User{2,name=Семен,surname=Петров},  
3=User{3,name=Алексей,surname=Давыдов}}
```



Если мы добавим еще одного пользователя, **id которого будет совпадать с id любого другого пользователя**, будет ли он добавлен в коллекцию TreeMap?

Добавим новый элемент user4

TreeMap не допускает хранение элементов с одинаковыми ключами, поэтому элемент с id, который был добавлен в коллекцию ранее, будет обновлен и мы получим результат работы программы, показанный справа.

```
public static void main(String[] args) {  
  
    // Создание коллекции  
    TreeMap<Integer, User> users = new TreeMap<>();  
  
    // Создание пользователей  
    User user1 = new User(1, "Иван", "Крайнов");  
    User user2 = new User(2, "Семен", "Петров");  
    User user3 = new User(3, "Алексей", "Давыдов");  
    User user4 = new User(1, "Николай", "Смирнов");  
  
    // Добавление пользователей в TreeMap  
    users.put(user3.id, user3);  
    users.put(user1.id, user1);  
    users.put(user2.id, user2);  
    users.put(user4.id, user4);  
  
    // Вывод пользователей  
    System.out.println(users);  
}
```

Результат работы программы:

```
{1=User{1,name=Николай,surname=Смирнов},  
2=User{2,name=Семен,surname=Петров},  
3=User{3,name=Алексей,surname=Давыдов}}
```



Вопрос

Что будет, если мы захотим добавить пользователя с ключом `null`?

Напишите в чат номер **одного** правильного ответа:

1. Будет добавлен, это обычная ситуация;
2. Произойдет ошибка компиляции;
3. Произойдет ошибка во время выполнения программы.

Ответ

Что будет, если мы заходим добавить пользователя с ключем null?

Ответ:

1. Будет добавлен, это обычная ситуация;
2. Произойдет ошибка компиляции;
- 3. Произойдет ошибка во время выполнения программы.**

Подробный ответ

Почему мы получим именно ошибку во время выполнения программы?

В **TreeSet** **нельзя** использовать **null** в качестве ключа, так как на нем нет возможности вызвать метод **compareTo**.

При использовании объектов в качестве ключей, отличных от строк и чисел, необходимо явно определять порядок сортировки этих объектов.

Для этого есть два способа:

- реализация интерфейса **Comparable**;
- создание класса **Comparator**;

Реализация интерфейса Comparable

Пример реализации интерфейса Comparable для класса UserId.

```
public class UserId implements Comparable<User> {  
    long id;  
    LocalDate created;  
  
    @Override  
    public int compareTo(User o) {  
        return Long.compare(this.id, o.id);  
    }  
}
```

Проблема, которую несет реализация интерфейса Comparable. Если потребуется изменение в сравнении объектов и их порядка сортировки, придется изменять класс и метод compareTo.

Интерфейс Comparable реализован у строк и чисел, потому что чаще всего мы ожидаем от них естественный порядок сортировки.

Реализация интерфейса `Comparator`

Класс, реализующий интерфейс `Comparator`, создается отдельно от класса, для которого он будет использоваться.

```
public class User implements Comparable<User> {  
    long id;  
    LocalDate created;  
  
    @Override  
    public int compareTo(User o) {  
        return Long.compare(this.id, o.id);  
    }  
}
```

```
public class UserIdComparator implements Comparator<User> {  
  
    @Override  
    public int compare(User o1, User o2) {  
        int last = Long.compare(o1.id, o2.id);  
        return last == 0 ? o1.created.compareTo(o2.created) : last;  
    }  
}
```

Если в конструктор `TreeMap` передать объект, реализующий интерфейс `Comparator`, то сортировка объектов будет происходить с использованием этого класса и его метода `compare`. Даже если у класса сравнения реализован метод `compareTo` интерфейса `Comparable`, он будет игнорироваться.

Скорость работы основных операций TreeMap

	containsKey	get	put	remove
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
HashMap	$O(1)$	$O(1)$	$O(1)$	$O(1)$

TreeMap гарантирует скорость доступа **$\log(n)$** для операций `containsKey`, `get`, `put` и `remove`.



Зачем нужен `TreeMap`,
кроме того, что он сортирует
данные по ключу?

Зачем еще нужен **TreeMap**?

1. Добавляет упорядоченное итерирование по ключу. Это нам необходимо, если нужно вывести элементы. Например, вызов `toString` на каждом `value` (варианты создания итератора: `keySet().iterator()` или `values().iterator()`).
2. Дает полезное свойство в отсортированной коллекции — возможность работать с диапазонами значений, используя методы **subSet** или **tailSet**.
[https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html#tailSet\(E\)](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html#tailSet(E))
3. Используется в ряде задач, где требуется гарантированная скорость доступа **$O(\log N)$** .
4. Используется в задачах, где нет возможности соблюсти контракт **HashCode** и **Equals**. Например, при использовании объектов с неизменяемым **HashCode**.

Структура данных TreeSet

Структура данных TreeSet

TreeSet — структура данных, которая хранит только уникальные элементы в упорядоченном виде. В своей реализации TreeSet использует TreeMap, только у этого TreeMap не используется поле value, а все элементы добавляются в качестве ключа.

Код из JDK:

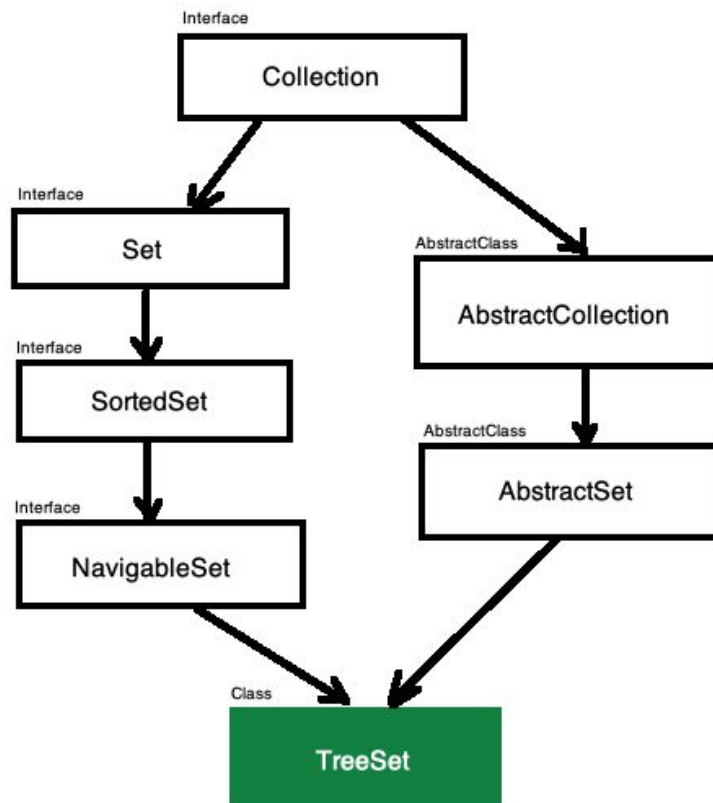
```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
{
    /**
     * The backing map.
     */
    private transient NavigableMap<E, Object> m;

    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    /**
     * Constructs a set backed by the specified navigable map.
     */
    TreeSet(NavigableMap<E, Object> m) {
        this.m = m;
    }
}
```

Устройство TreeSet

Класс **TreeSet** наследует **AbstractSet** класс и реализует интерфейс **NavigableSet**.





Методы коллекции `TreeSet`

Коллекция `TreeSet` наследуется от абстрактного класса `NavigableSet` и реализует методы интерфейса `SortedSet`.

Методы интерфейса `SortedSet`

- **`comparator()`** — возвращает используемый компаратор, если он был задан, либо `null`, если в коллекции используется natural ordering.
- **`E first()`** — возвращает первый элемент `Set`.
- **`E last()`** — возвращает последний элемент `Set`.
- **`SortedSet headSet(E end)`** — возвращает объект `SortedSet`, который содержит все элементы первичного `Set` до элемента `end`.
- **`SortedSet subSet(E start, E end)`** — возвращает объект `SortedSet`, который содержит все элементы первичного `Set` между элементами `start` и `end`.
- **`SortedSet tailSet(E start)`** — возвращает объект `SortedSet`, который содержит все элементы первичного `Set`, начиная с элемента `tail`.

Методы абстрактного класса `NavigableSet`

- **`E ceiling(E obj)`** — ищет в `Set` наименьший элемент `e`, который больше или равен `obj` ($e \geq obj$). Если такой элемент найден, то он возвращается в качестве результата, иначе возвращается `null`.
- **`E floor(E obj)`** — ищет в наборе наибольший элемент `e`, который меньше или равен элементу `obj` ($e \leq obj$). Если такой элемент найден, то он возвращается в качестве результата. Иначе возвращается `null`.
- **`E higher(E obj)`** — ищет в наборе наименьший элемент `e`, который больше элемента `obj` ($e > obj$). Если такой элемент найден, то он возвращается в качестве результата, иначе возвращается `null`.
- **`E lower(E obj)`** — ищет в наборе наибольший элемент `e`, который меньше элемента `obj` ($e < obj$). Если такой элемент найден, то он возвращается в качестве результата, иначе возвращается `null`.

Методы абстрактного класса `NavigableSet`

- **`E pollFirst()`** — возвращает первый элемент и удаляет его из `Set`, или возвращает `null` в случае, если сет пустой.
- **`E pollLast()`** — возвращает последний элемент и удаляет его из `Set`, или возвращает `null` в случае если сет пустой.
- **`NavigableSet descendingSet()`** — возвращает объект `NavigableSet`, который содержит все элементы первичного набора `NavigableSet` в обратном порядке.

[Подробнее >>](#)

Конструктор `TreeSet` со своим компаратором

Помимо конструктора без параметра, можно создать объект `TreeSet` с конструктором, который позволяет определить порядок сортировки элементов, используя `Comparator` (все как у `TreeMap`).

```
// Создание пустого древовидного набора с сортировкой согласно естественному
// упорядочиванию его элементов
TreeSet()

// Создание древовидного набора, содержащего элементы в указанном наборе,
// с сортировкой согласно естественному упорядочиванию его элементов.
TreeSet(Collection<? extends E> c)

// Создание пустого древовидного набора, с сортировкой согласно comparator
TreeSet(Comparator<? super E> comparator)

// Создание древовидного набора, содержащего те же самые элементы и
// использующего
// то же самое упорядочивание в качестве указанного сортированного набора
TreeSet(SortedSet<E> s)
```

Отличия между `TreeSet` и `HashSet`

- `HashSet` не поддерживает никакого порядка хранения элементов, `TreeSet` применяет натуральную сортировку к элементам.
- `HashSet` может хранить `null` объекты, `TreeSet` не может.
- `HashSet` для сравнения элементов использует `equals()` и `hashCode()`, `TreeSet` использует `compare()` или `compareTo()`.
- `HashSet` обеспечивает лучшую производительность — константную временную производительность для большинства операций `add()`, `remove()` и `contains()`, `TreeSet` обеспечивает $\log(n)$.
- `TreeSet` благодаря методам `pollFirst()`, `pollLast()`, `first()`, `last()`, `ceiling()`, `lower()` можно использовать более гибко.

Применение TreeSet

TreeSet необходимо использовать:

- когда требуется эффективный способ сортировать по какому-то признаку или нескольким признакам большое количество уникальных элементов, но следует держать в уме, что TreeSet немного медленнее HashMap;
- если нет возможности задать «хорошую» хэш-функцию распределения объектов, то TreeSet может оказаться даже эффективнее;
- также полезно использовать методы TreeSet: headSet, tailSet, subSet при операциях поиска или выделения подмножества над отсортированным множеством.

Применение TreeSet

Сначала рассмотрим пример с простыми строками, по умолчанию они будут сортироваться по алфавиту.

```
import java.util.*;

public class Program{

    public static void main(String[] args) {

        // каталог книг в онлайн-магазине
        TreeSet<String> catalog = new TreeSet<String>();

        // добавим в список ряд элементов
        catalog.add("1984");
        catalog.add("Шантарам");
        catalog.add("Властелин Колец");
        catalog.add("Гарри Поттер");
        catalog.add("Три товарища");

        System.out.println(catalog.first());
        // получим первый – самый меньший элемент – 1984

        System.out.println(catalog.last());
        // получим последний – самый больший элемент – Шантарам
```

Применение TreeSet

```
// получим поднабор от одного элемента до другого
SortedSet<String> set = catalog.subSet("Властелин Колец",
"Шантарам");
System.out.println(set);
// [Властелин Колец, Гарри Поттер, Три товарища]

// элемент из набора, который больше текущего
String greater = catalog.higher("1984");
System.out.println(greater);
// [Властелин Колец]

// элемент из набора, который меньше текущего
String lower = catalog.lower("Три товарища");
System.out.println(lower);
// [Гарри Поттер]

// возвращаем набор в обратном порядке
NavigableSet<String> navSet = catalog.descendingSet();
System.out.println(navSet);
// [Шантарам, Три товарища, Гарри Поттер, Властелин Колец, 1984]
```

Применение TreeSet

```
// возвращаем набор, в котором все элементы меньше текущего
SortedSet<String> setLower = catalog.headSet("Три товарища");
System.out.println(setLower);
// [1984, Властелин Колец, Гарри Поттер]

// возвращаем набор, в котором все элементы больше текущего
SortedSet<String> setGreater = catalog.tailSet("Властелин
Колец");
System.out.println(setGreater);
// [Властелин Колец, Гарри Поттер, Три товарища, Шантарам]

}
```


Применение TreeSet

Рассмотрим пример с комплексным объектом:

```
class Person{

    private String name;
    private String surname;
    private int age;

    Person(String name, String surname, int age){

        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    String getName(){return name;}
    String getSurname(){return surname;}
    int getAge(){return age;}
}
```

Применение TreeSet

Мы не можем просто создать объект `TreeSet` и добавить в него объект `Person`. В случае добавления объектов, `TreeSet` не будет знать, как их сравнивать, и следующий кусок кода работать не будет:

```
TreeSet<Person> people = new TreeSet<Person>();  
people.add(new Person("Tom", "Morris", 29));
```

Получим `java.lang.ClassCastException`.

Применение TreeSet

Создадим **Comparator<Person>**, чтобы получить возможность добавлять элементы в TreeSet.

```
class Person implements Comparable<Person>{

    private String name;
    private String surname;
    private int age;

    ....

    public int compareTo(Person p){

        return name.compareTo(p.getName());
    }
}
```

Применение TreeSet

Добавим Comparator в TreeSet:

```
PersonComparator pcomp = new PersonComparator();
TreeSet<Person> people = new TreeSet<Person>(pcomp);
people.add(new Person("Tom", "Morris", 29));
people.add(new Person("Nick", "Harris", 16));
people.add(new Person("Alex", "Ivanov", 22));
people.add(new Person("Bill", "Kim", 45));
for (Person p : people) {
    System.out.println(p.getSurname());
}
// [Harris, Ivanov, Kim, Morris]
```

Для создания TreeSet здесь используется одна из версий конструктора, которая в качестве параметра принимает компаратор. Теперь вне зависимости от того, реализован ли в классе Person интерфейс Comparable, логика сравнения и сортировки будет использоваться та, которая определена в классе компаратора.

Сортировка по нескольким критериям

Начиная с JDK 8, в механизм работы компараторов были внесены дополнения. Теперь мы можем применять сразу несколько компараторов по принципу приоритета. Например, отсортируем пользователей по имени и по возрасту (от старшего к младшему):

```
TreeSet<Person> people = new
TreeSet<>(Comparator.comparing(Person::getName)
    .thenComparing(Person::getAge, Comparator.reverseOrder()));


people.add(new Person("Tom", "Morris", 29));
people.add(new Person("Nick", "Harris", 16));
people.add(new Person("Alex", "Ivanov", 22));
people.add(new Person("Tom", "Kim", 45));
for (Person p : people) {
    System.out.println(p.getName() + " " + p.getSurname());
}
// Alex Ivanov
// Nick Harris
// Tom Kim
// Tom Morris
```

Создавать **comparator** с помощью отдельного класса или использовать **Comparator.comparing** остается на усмотрение разработчика.


Дополнительные материалы

[Java собеседование. Коллекции](#)

(вам нужны пункты с 15 по 17)

Хабр |  [КАК СТАТЬ АВТОРОМ](#)

[Все потоки](#) [Разработка](#) [Администрирование](#) [Дизайн](#) [Менеджмент](#) [Маркетинг](#) [Научпоп](#)

 **sphinks** 11 декабря 2012 в 16:34

Java собеседование. Коллекции

Java, Алгоритмы



Итоги

Итоги

- Рассмотрели коллекцию `TreeMap`;
- Рассмотрели коллекцию `TreeSet`;
- Сравнили `HashSet`, `HashMap` с коллекциями основанными на деревьях `TreeMap`, `TreeSet`;
- Научились писать компараторы.



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

Юрий Пеньков