Java集合

知识点

- ✓ Java数据结构概述
- ✓ Java集合框架概述
- ✓ 核心接口
- ✓ 主要实现类
- ✓ 集合框架性能分析

数组（Array）

- 数组（Arrays）是一种基本的数据结构，可以存储固定大小的相同类型的元素。

- int[] array = new int[5]
- String[] array = new String[5]

- 特点： 固定大小，存储相同类型的元素（基本类型值或对象）
- 优点： 随机访问元素效率高（存储和获取）
- 缺点： 大小固定，插入和删除元素较慢（copy）

列表（List）

- 按照一定的线性顺序，排列而成的数据项的集合
- 数组 vs. 列表
  - 列表是动态的，可以随时添加、删除或更改元素
  - 列表存的是对象（int vs. Integer）

包装类：对基本数据类型包装，提供更多的方法和属性。Integer.Max_VALUE

- List<String> arrayList = new ArrayList<>();
- List<Integer> linkedList = new LinkedList<>();

ArrayList:

- 特点： 动态数组，可变大小
- 优点： 高效的随机访问和快速尾部插入、内存小
- 缺点： 中间插入和删除相对较慢

LinkedList:

- 特点： 双向链表，元素之间通过指针连接
- 优点： 插入和删除元素高效、内存大
- 缺点： 随机访问相对较慢

集合（Set）

集合（Sets）用于存储不重复的元素，常见的实现有 HashSet 和 TreeSet。

- Set<String> hashSet = new HashSet<>();
- Set<Integer> treeSet = new TreeSet<>();

HashSet:

- 特点： 无序集合
- 优点： 高效的查找和插入操作。
- 缺点： 不保证顺序。

TreeSet:

- 特点：TreeSet 是有序集合，底层基于红黑树实现，不允许重复元素。
- 优点： 提供自动排序功能，适用于需要按顺序存储元素的场景。
- 缺点： 性能相对较差，不允许插入 null 元素。

集合（Set）

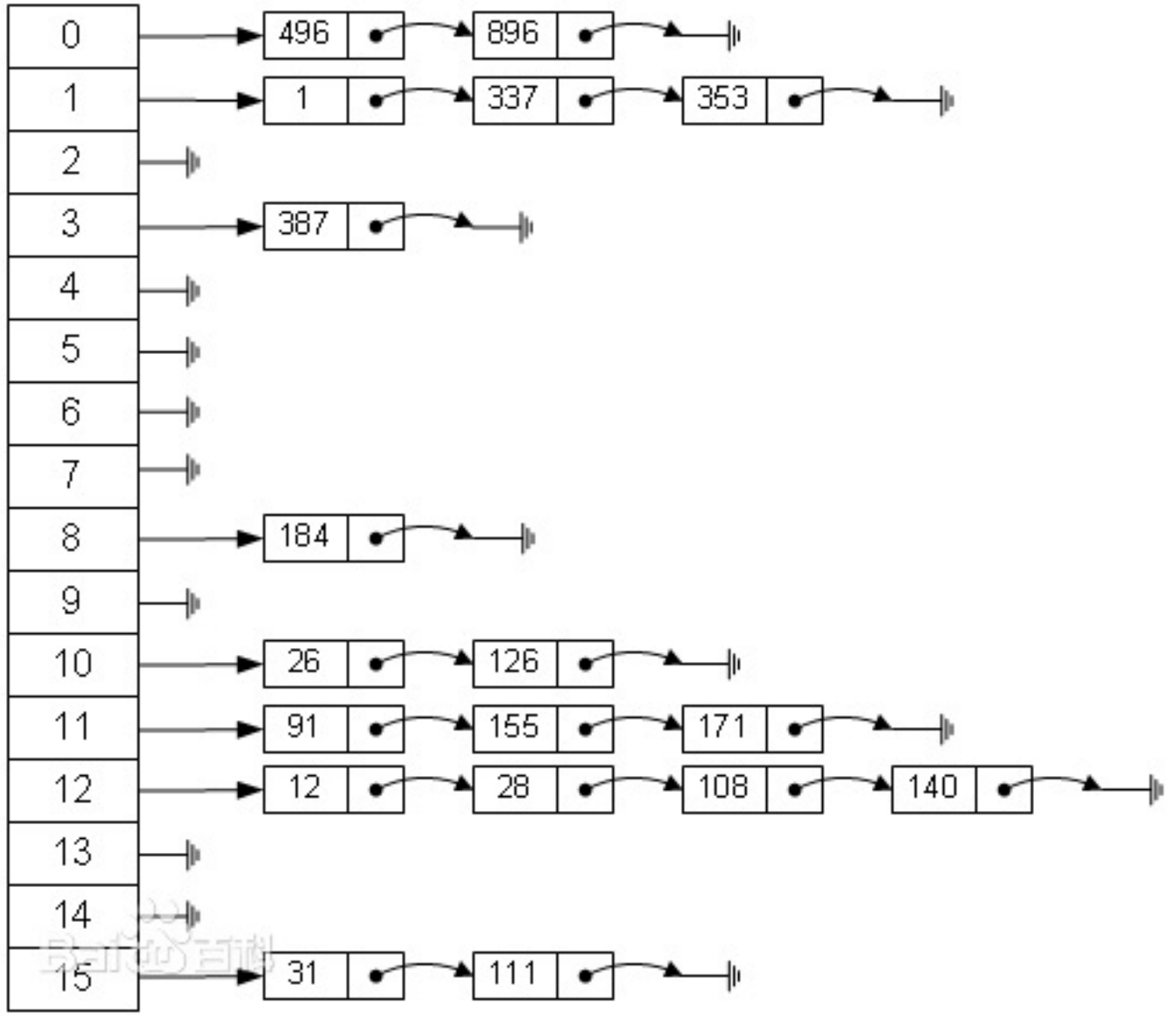
哈希表

用哈希函数计算哈希值确定存储位置

index = key % capacity

index = A * key + B

用链表解决冲突



映射（Map）

映射用于存储键值对<key, value>，常见的实现有 HashMap 和 TreeMap。

- Map<String, Integer> hashMap = new HashMap<>();
- Map<String, Integer> treeMap = new TreeMap<>();

HashMap:

- 特点： 基于哈希表实现的键值对存储结构。
- 优点： 高效的查找、插入和删除操作。
- 缺点： 无序，不保证顺序。

TreeMap:

- 特点： 基于红黑树实现的有序键值对存储结构。
- 优点： 有序，支持按照键的顺序遍历。
- 缺点： 插入和删除相对较慢。

栈（Stack）

栈（Stack）是一种线性数据结构，它按照后进先出（Last In, First Out，LIFO）的原则管理元素。在栈中，新元素被添加到栈的顶部，而只能从栈的顶部移除元素。这就意味着最后添加的元素是第一个被移除的。

```
Stack<Integer> stack = new Stack<>();
```

Stack 类

- 特点： 代表一个栈，通常按照后进先出（LIFO）的顺序操作元素。

队列（Queue）

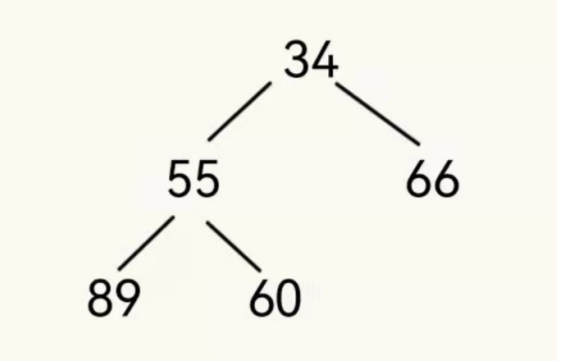
队列（Queue）遵循先进先出（FIFO）原则，可以使用数组或者链表去实现，常见的实现有 LinkedList 和 PriorityQueue。

```
Queue<String> queue = new LinkedList<>();
```

Queue 接口:

- 特点： 代表一个队列，通常按照先进先出（FIFO）的顺序操作元素。
- 实现类： LinkedList, PriorityQueue, ArrayDeque。

堆（Heap）



堆中某个结点的值总是不大于或不小于其父结点的值

堆总是一棵完全二叉树。

leftchild = parent * 2 + 1;
rightchild = parent * 2 + 2;
parent = (child - 1) / 2

```
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
```

内存中的堆栈（广义）

栈（stack）：由编译器自动分配和释放，存放函数的参数、局部变量、临时变量、函数返回地址等；

堆（heap）：一般由程序员分配和释放（new），如果没有手动释放，在程序结束时可能由操作系统自动释放

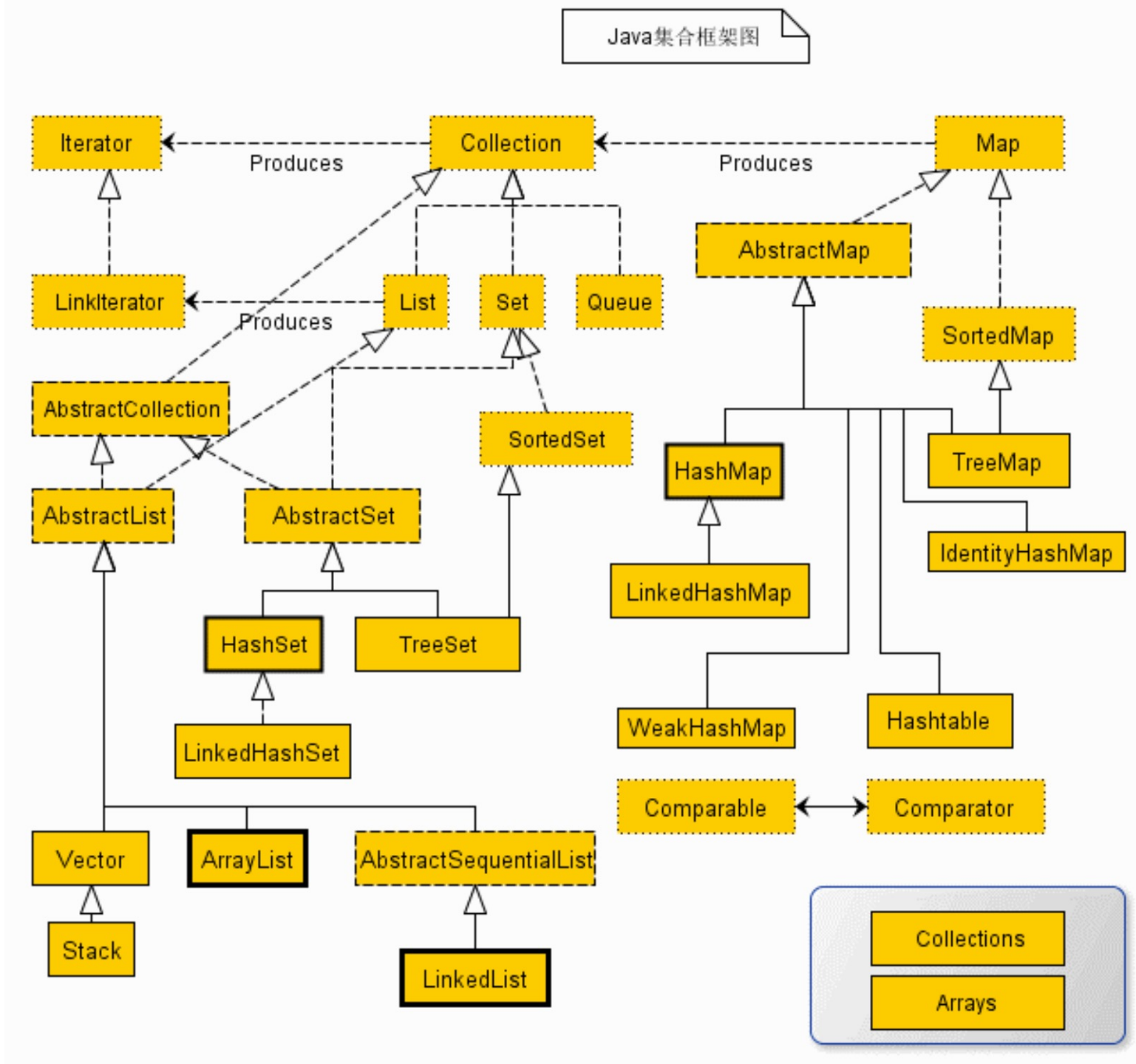
树（Tree）

Java 提供了 TreeNode 类型，可以用于构建二叉树等数据结构。

```
class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;
        TreeNode(int x) { val = x; }
}
```

图（Graph）

图的表示通常需要自定义数据结构或使用图库，Java 没有内建的图类。

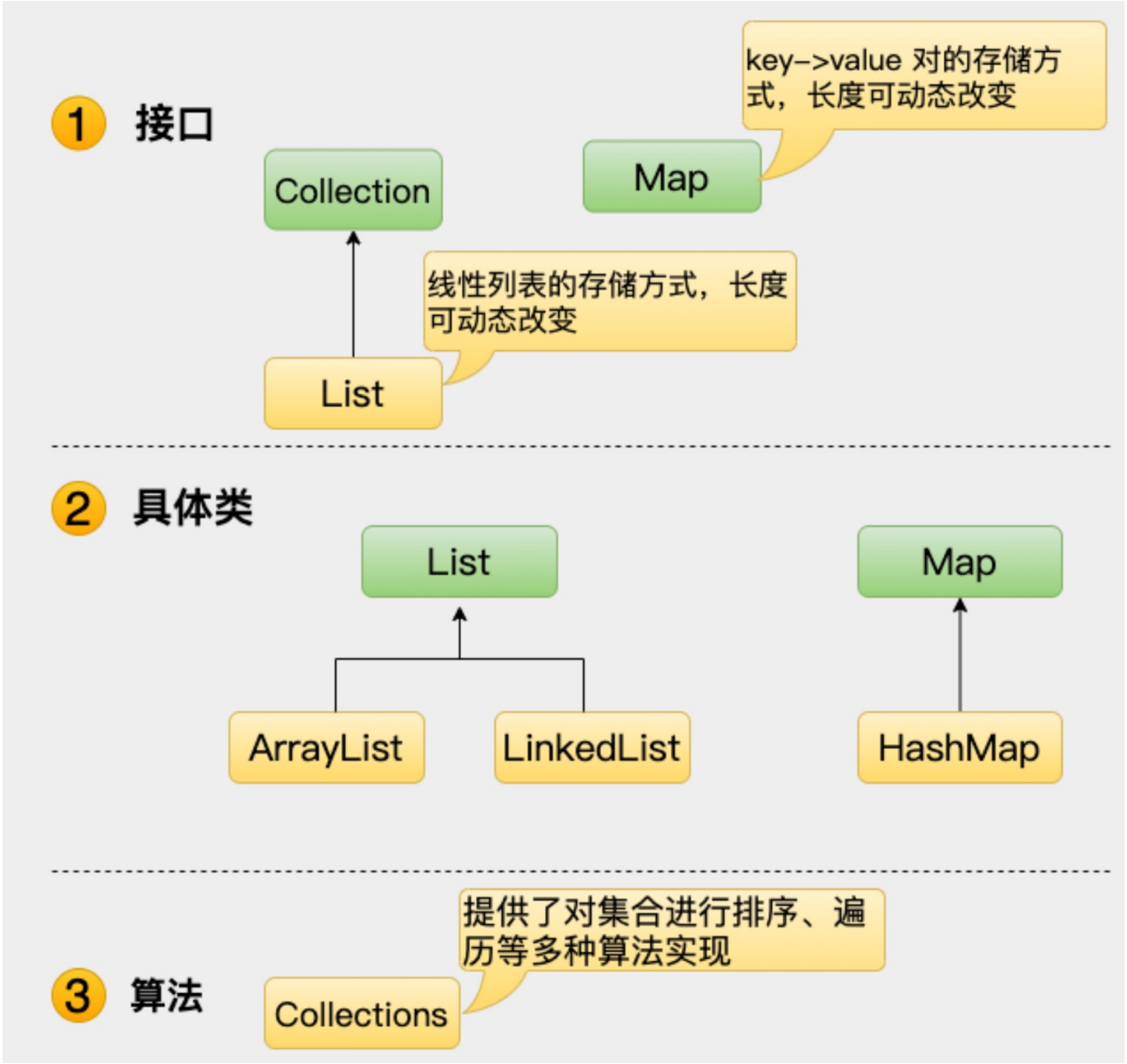

Java集合框架图
Iterator
Produces
Collection
Produces
Map
LinkIterator
Produces
List
Set
Queue
AbstractMap
SortedMap
AbstractCollection
SortedSet
HashMap
TreeMap
AbstractList
AbstractSet
IdentityHashMap
LinkedHashMap
HashSet
TreeSet
WeakHashMap
Hashtable
LinkedHashSet
Comparable
Comparator
Vector
ArrayList
AbstractSequentialList
Collections
Arrays
Stack
LinkedList

接口、实现类、算法

- 接口：是代表集合的抽象数据类型。例如 Collection、List、Set、Map 等。之所以定义多个接口，是为了以不同的方式操作集合对象

- 实现（类）：是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构，例如：ArrayList、LinkedList、HashSet、HashMap。

- 算法：是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序，这些算法实现了多态，那是因为相同的方法可以在相似的接口上有着不同的实现。


1 接口
Collection
Map
key–>value 对的存储方式，长度可动态改变
线性列表的存储方式，长度可动态改变
List
2 具体类
List
Map
ArrayList
LinkedList
HashMap
3 算法
提供了对集合进行排序、遍历等多种算法实现
Collections

核心接口

1. Collection：集合根接口
2. List：有序集合，允许重复元素（如ArrayList、LinkedList）
3. Set：不允许重复元素的集合（如HashSet、TreeSet）
4. Queue：支持FIFO（先进先出）原则的集合（如LinkedList、PriorityQueue）
5. Map：键值对映射（如HashMap、TreeMap）

主要实现类

- ArrayList：动态数组实现
- LinkedList：双向链表实现
- HashSet：基于哈希表的集合
- TreeSet：基于红黑树的有序集合
- HashMap：基于哈希表的键值对映射
- TreeMap：基于红黑树的有序键值对映射

ArrayList

- 特点：基于动态数组实现
- 性能：随机访问快，插入和删除较慢（需要移动元素）
- 方法：
  - add(E e): 添加元素
  - get(int index): 获取指定位置的元素
  - set(int index, E element): 替换指定位置的元素
  - remove(int index): 删除指定位置的元素
  - size(): 获取元素数量
  - isEmpty(): 检查是否为空
  - iterator(): 返回Iterator用于遍历

LinkedList

- 特点：基于链表实现
- 性能：插入和删除快
- 方法：
  - add(E e): 添加元素
  - get(int index): 获取指定位置的元素
  - set(int index, E element): 替换指定位置的元素
  - remove(int index): 删除指定位置的元素
  - size(): 获取元素数量
  - isEmpty(): 检查是否为空
  - iterator(): 返回Iterator用于遍历

ArrayList vs. LinkedList

- ArrayList：
  - 优点：随机访问快，内存连续-----时间复杂度O(1)
  - 缺点：插入和删除慢（需要移动元素）-----时间复杂度O(N)

- LinkedList：
  - 优点：插入和删除快（只需改变引用）----时间复杂度O(1)
  - 缺点：随机访问慢，内存分散----时间复杂度O(N)

HashSet

- 特点：基于哈希表实现，不保证顺序
- 性能：常数时间复杂度的增删查操作
- 方法：
  - add(E e): 添加元素
  - remove(Object o): 删除元素
  - contains(Object o): 检查是否包含某元素
  - size(): 获取元素数量
  - isEmpty(): 检查是否为空
  - iterator(): 返回Iterator用于遍历

TreeSet

- 特点：基于红黑树实现，保证元素有序
- 性能：对数时间复杂度的增删查操作
- 方法：
  - add(E e): 添加元素
  - remove(Object o): 删除元素
  - contains(Object o): 检查是否包含某元素
  - size(): 获取元素数量
  - isEmpty(): 检查是否为空
  - iterator(): 返回Iterator用于遍历
  - first(): 返回第一个（最小）元素
  - last(): 返回最后一个（最大）元素
  - headSet(E toElement): 返回小于toElement的子集
  - tailSet(E fromElement): 返回大于或等于fromElement的子集

性能对比

- 时间复杂度：
  - HashSet的增删查是O(1)
  - TreeSet的增删查是O(log n)
- 内存消耗：
  - HashSet内存使用相对较低
  - TreeSet由于树结构，内存使用相对较高

Stack

- Stack 类实现了后进先出 (LIFO) 的数据结构
- 方法：
  - boolean empty()：测试栈是否为空
  - E push(E item)：将项目推入栈的顶部
  - E pop()：移除并返回栈顶部的项目
  - E peek()：查看栈顶部的对象，但不移除它
  - int search(Object o): 返回对象在栈中的位置，以 1 为基数（即返回距离栈顶最近的位置）。如果对象在栈中，则返回从栈顶开始的位置；否则返回 -1。

Queue

- Queue 接口表示一个队列，它通常遵循先进先出（FIFO）的顺序
- 方法：
  - boolean add(E e): 将指定的元素插入此队列（如果立即可行且不会违反容量限制），当使用容量受限的队列时，此方法通常优于 offer。
  - boolean offer(E e): 将指定的元素插入此队列（如果立即可行且不会违反容量限制），如果当前没有可用空间，则返回 false 而不是抛出异常。
  - E remove(): 获取并移除此队列的头
  - E poll(): 获取并移除此队列的头
  - E peek(): 获取但不移除此队列的头

Queue

- Map 接口是用来存储键值对的集合
- 方法：
  - V put(K key, V value): 将指定的值与指定的键相关联，并将其插入到 Map 中。如果 Map 中已经存在相同的键，则会替换原有的值，并返回被替换的值
  - V get(Object key): 返回指定键所映射的值
  - boolean containsKey(Object key): 如果此 Map 包含指定键的映射关系，则返回 true
  - boolean containsValue(Object value): 如果此 Map 将一个或多个键映射到指定值，则返回 true
  - V remove(Object key): 如果存在一个键的映射关系，则将其从 Map 中移除
  - int size(): 返回此 Map 中的键值对数量

Iterator（迭代器）

- Iterator方法：
  - next() - 返回迭代器的下一个元素，并将迭代器的指针移到下一个位置
  - hasNext() - 用于判断集合中是否还有下一个元素可以访问
  - remove() - 从集合中删除迭代器最后访问的元素

Iterator（迭代器）

```java
import java.util.ArrayList;
import java.util.Iterator;

public class RunoobTest {
    public static void main(String[] args) {

        // 创建集合
        ArrayList<String> sites = new ArrayList<String>();
        sites.add("Google");
        sites.add("Runoob");
        sites.add("Taobao");
        sites.add("Zhihu");

        // 获取迭代器
        Iterator<String> it = sites.iterator();

        // 输出集合中的第一个元素
        System.out.println(it.next());
    }
}
```

Collections

Java中的Collections是一个提供静态方法的实用程序类，用于操作或返回集合（特别是列表和集）

1. sort()

sort()方法用于对列表进行排序。它使用Comparable接口提供的compareTo()方法对元素进行排序，或者可以传入一个自定义的Comparator来定义排序逻辑。

```
public class SortExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Zoe");
        names.add("Bob");
        names.add("Alice");
        names.add("Charlie");

        // 使用自然排序（字母顺序）
        Collections.sort(names);
        System.out.println("Sorted list in natural order: " + names);

        // 使用自定义排序（按长度）
        Collections.sort(names, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
        System.out.println("Sorted list by length: " + names);
    }
```

Collections

binarySearch()方法在已排序的列表中执行二分查找，返回指定元素的索引。需要注意的是，这个方法假定列表已经按照升序排列。

```
public class BinarySearchExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        numbers.add(5);
        Collections.sort(numbers); // 确保列表已排序
        int index = Collections.binarySearch(numbers, 3); // 查找元素3的索引
        System.out.println("Index of 3: " + index); // 应该输出2，因为索引是从0开始的
    }
}
```

Collections

shuffle()方法用于随机重新排列列表中的元素。这对于创建随机化的数据集非常有用。

```
public class ShuffleExample {
    public static void main(String[] args) {
        List<String> cards = new ArrayList<>();
        cards.add("Heart Ace");
        cards.add("Heart King");
        cards.add("Spade Queen");
        cards.add("Club Jack");
        System.out.println("Before shuffle: " + cards);
        Collections.shuffle(cards); // 打乱顺序
        System.out.println("After shuffle: " + cards); // 每次运行结果可能不同，因为顺序是随机的
    }
}
```

Collections

reverse()方法用于反转列表中元素的顺序。这个方法会直接修改传入的列表，不会返回新的列表。

```
public class ReverseExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("Before reverse: " + numbers); // 输出[1, 2, 3, 4]
        Collections.reverse(numbers); // 反转列表顺序
        System.out.println("After reverse: " + numbers); // 输出[4, 3, 2, 1]
    }
}
```