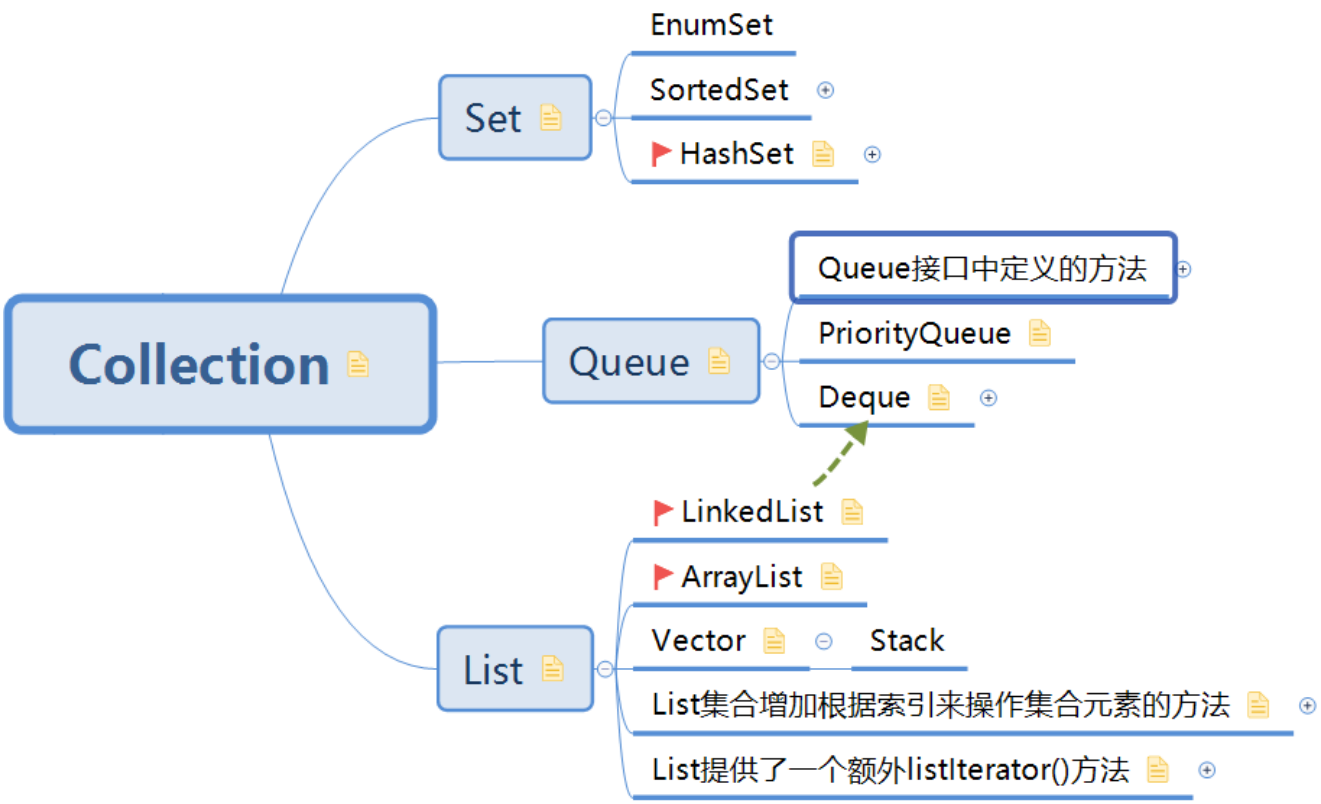
集合	扩容方式	内部实现
ArrayList	1.5倍扩容=原来的容量*1.5； 当调用add方法时发现原始数组容量不够，则扩容。 如果在创建的时候没有指定初始容量，默认容量为10。	Object[] elementData
Vector	Vector扩容方式可以自己在创建Vector时指定一次扩容的增加量，若未指定则扩容为原来容量的2倍； 当调用add方法时发现原始数组容量不够，则扩容； 如果创建时没有指定初始容量，默认为10；	Object[] elementData
LinkedList	无容量限制	双向链表<pre, next>
HashSet	新容量为原来的 2 倍，且保证容量为2^n	内部由HashMap实现
PriorityQueue	当存储的数据量达到数组的长度时，再插入数据时就需要扩容了。 当原始容量少于64时，则扩容为（oldCapacity*2 + 2）； 否则，扩容为1.5 * oldCapacity。默认初始容量为11。	Object[] queue <小顶堆>
ArrayDeque	在每次offer()一个元素之后就会检查，当前数组是否已经填满，若已经填满，则扩容为原始容量的2倍。	transient Object[] elements <循环队列>



一、List

List集合代表一个元素有序、可重复的集合，集合中每个元素都有其对应的顺序索引。List集合允许使用重复元素，可以通过索引来访问指定位置的集合元素。索引从0开始。

对于例子：

```
class A{
    public boolean equals(Object obj){
        return true;
    }
}

List books = new ArrayList();
books.add("a");
books.add("b");
books.add("c");
books.remove(new A());
```

对于执行的remove操作，会调用A的equals方法一次与集合元素进行比较，如果该equals()方法以某个集合元素作为参数是返回true，list将会删除该元素——A类重写了equals()方法，总是返回true，因此每次从list中删除a对象时，总是删除list的第一个元素。

固定长度的List：

Arrays工具类里提供了asList(Object... a)方法，该方法可以把一个数组或指定个数的对象转换成一个List集合，这个List集合既不是ArrayList实现类的实例，也不是Vector实现类的实例，而是Arrays的内部类ArrayList的实例。Arrays.ArrayList是一个固定长度的List集合，程序只能遍历访问该集合里的元素，不可增加、删除该集合里的元素。
List books = Arrays.asList("a","b","c","d");

关于使用List集合的建议：

1）如果需要遍历List集合元素，ArrayList、Vector使用随机访问方法（get）来遍历集合元素。对于LinkedList则应该使用Iterator迭代器来遍历

2）如果需要经常执行插入、删除操作来改变包括大量数据的List集合的大小，可以考虑使用LinkedList集合。ArrayList、Vector集合可能需要经常重新分配内部数组的大小，效果可能较差。

3）如果有多个线程需要同时访问List集合中的元素，开发者可考虑使用Collections将集合包装成线程安全的集合。

1. ArrayList

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{...}
```

使用无参构造函数时，在第一次调用add方法时，会使用默认数组长度（DEFAULT_CAPACITY=10）创建数组。扩容方式是按照1.5倍的方式进行扩容。

ArrayList是线程不安全的，可以使用Collections.synchronizedList(new ArrayList())来保证线程安全。 Vector是线程安全的，无序程序保证该集合的同步性。

在我们学数据结构的时候就知道了线性表的顺序存储，插入删除元素的时间复杂度为**O（n）**,求表长以及增加元素，取第 i 元素的时间复杂度为**O（1）**

ArrayList 继承了AbstractList，实现了List。它是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

ArrayList 实现了**RandomAccess 接口**， RandomAccess 是一个标志接口，表明实现这个这个接口的 List 集合是支持**快速随机访问**的。在 ArrayList 中，我们即可以通过元素的序号快速获取元素对象，这就是快速随机访问。

ArrayList 实现了**Cloneable 接口**，即覆盖了函数 clone()，**能被克隆。**

ArrayList 实现**java.io.Serializable 接口**，这意味着ArrayList**支持序列化**，**能通过序列化去传输。**

和 Vector 不同，**ArrayList 中的操作不是线程安全的**！所以，建议在单线程中才使用 ArrayList，而在多线程中可以选择 Vector 或者 CopyOnWriteArrayList。

（1）ArrayList构造函数

```
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        // 根据初始参数创建对象数组
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    }
}
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
public ArrayList(Collection<? extends E> c) {
    elementData = c.toArray();
    if ((size = elementData.length) != 0) {
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    } else {
        // replace with empty array.
        this.elementData = EMPTY_ELEMENTDATA;
    }
}
```

（2）add()方法

在add的时候，如果超过了已有数组的长度，那么需要将数组长度扩容为原来数组长度的1.5倍。

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

```
6
7   private void ensureCapacityInternal(int minCapacity) {
8       if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
9           minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
10      }
11
12      ensureExplicitCapacity(minCapacity);
13  }
14
15  private void ensureExplicitCapacity(int minCapacity) {
16      modCount++;
17      // minCapacity表示的是加入一个新元素之后的最小长度，如果比已有数组长度要大，就需要增加长度
18      // overflow-conscious code
19      if (minCapacity - elementData.length > 0)
20          grow(minCapacity);
21  }
22
23  private void grow(int minCapacity) {
24      // overflow-conscious code
25      int oldCapacity = elementData.length;
26      // 扩容为1.5倍
27      int newCapacity = oldCapacity + (oldCapacity >> 1);
28      if (newCapacity - minCapacity < 0)
29          newCapacity = minCapacity;
30      if (newCapacity - MAX_ARRAY_SIZE > 0)
31          newCapacity = hugeCapacity(minCapacity);
32      // minCapacity is usually close to size, so this is a win:
33      elementData = Arrays.copyOf(elementData, newCapacity);
34  }
35
36  private static int hugeCapacity(int minCapacity) {
37      if (minCapacity < 0) // overflow
38          throw new OutOfMemoryError();
39      return (minCapacity > MAX_ARRAY_SIZE) ?
40          Integer.MAX_VALUE :
41      MAX_ARRAY_SIZE;
42  }
43
44  // 另外一个add方法，在指定位置添加元素
45  public void add(int index, E element) {
46      // 如果超出了数组的长度范围（范围为[0,size]）会抛出异常，IndexOutOfBoundsException
47      rangeCheckForAdd(index);
48
49      ensureCapacityInternal(size + 1);  // Increments modCount!!
50      // 拷贝index以及以后的到index+1位置
51      System.arraycopy(elementData, index, elementData, index + 1,
52                       size - index);
53      elementData[index] = element;
54      size++;
55  }
```

(3) remove()方法

```java
/**
* 删除该列表中指定位置的元素。 将任何后续元素移动到左侧（从其索引中减去一个元素）。
*/
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        // 将index+1以后的数据拷贝到index位置，覆盖原有数据
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
}
/**
* 从列表中删除指定元素的第一个出现（如果存在）。 如果列表不包含该元素，则它不会更改。
* 返回true，如果此列表包含指定的元素
*/
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
            }
    }
    return false;
}
/*
* Private remove method that skips bounds checking and does not
* return the value removed.
*/
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    // 将size处的设置为null
    elementData[--size] = null; // clear to let GC do its work
}
```

(4) ArrayList与Vector的对比

- Vector是线程安全的，所有方法都是同步的，可以由多个线程安全地访问一个Vector对象，但代码在同步操作上会耗费大量时间。ArrayList不是线程安全的，在不需要同步的情况下，推荐使用ArrayList，它的性能更好。
- 两者的扩容方式不同ArrayList是按照1.5倍的方式扩容，Vector可以按照用户设定的方式扩容，默认按照2倍进行扩容。

(5) ArrayList和LinkedList的对比

- 实现机制不同。ArrayList内部由数组实现，当容量达到数组容量上限需要扩容；LinkedList内部由双向链表实现，无具体的长度限制。
- 性能不同。ArrayList在随机访问上的性能要更好，但在插入、删除元素时LinkedList性能更佳。
- 他们都不是线程安全的，如果要保证线程安全需要使用Collections.synchronizedList(...)。

2.Vector

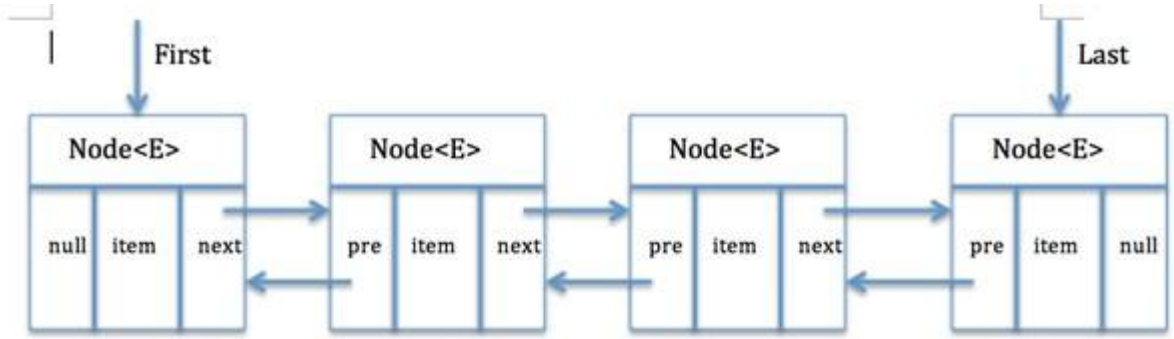
Vector有一个子类Stack，可以实现进栈(push(Object item))、出栈（pop()）、获得第一个元素（peak()）。但是它线程安全，性能较差，如果要用到"栈"这种数据结构，可以考虑使用ArrayDeque。

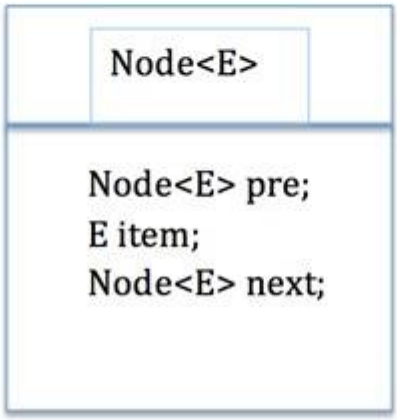
3.LinkedList

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{ ... }
```

LinkedList是一个实现了List接口和Deque接口的双端链表。 LinkedList底层的链表结构使它支持高效的插入和删除操作，另外它实现了Deque接口，使得LinkedList类也具有队列的特性;LinkedList不是线程安全的，如果想使LinkedList变成线程安全的，可以调用静态类Collections类中的synchronizedList方法：

```
List list=Collections.synchronizedList(new LinkedList(...));
```





LinkedList维护头结点first、尾结点last和size。

LinkedList是List接口的实现类，这意味着它是一个List集合，可以根据索引来随机访问集合中的元素。除此之外，**LinkedList还实现了Deque接口，可以被当成双端队列来使用，因此既可以当做栈，也可以当做队列使用。**

LinkedList与ArrayList、ArrayDeque的实现机制完全不同。前者内部以链表的形式来保存集合中的元素，因此随机访问元素时性能较差，但**在插入、删除元素时性能较出色。**后者内部以数组的形式来保存集合中的元素，使用随机访问的性能比使用Iterator迭代访问的性能要好，因为随机访问会被映射成对数组元素的访问。

（1）LinkedList构造方法

```
public LinkedList() {
}

public LinkedList(Collection<? extends E> c) {
    this();
    addAll(c);
}
private static class Node<E> {
    E item;//节点值
    Node<E> next;//后继节点
    Node<E> prev;//前驱节点

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
```

```
        this.prev = prev;
    }
}
```

(2) add()方法

```
public void addFirst(E e) {
    linkFirst(e);
}
/**
* 将节点添加到LinkedList头
*/
private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
    size++;
    modCount++;
}

public void addLast(E e) {
    linkLast(e);
}
/**
* 将节点添加到LinkedList尾
*/
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
public boolean add(E e) {
    linkLast(e);
    return true;
}
```

(3) remove()方法

```
/**
* 注意下面的removeFirst和removeLast方法在链表为空时，会抛出异常。
*/
public E removeFirst() {
```

```
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return unlinkFirst(f);
}
public E removeLast() {
    final Node<E> l = last;
    if (l == null)
        throw new NoSuchElementException();
    return unlinkLast(l);
}
public E remove() {
    return removeFirst();
}
public boolean remove(Object o) {
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x);
                return true;
            }
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item)) {
                unlink(x);
                return true;
            }
        }
    }
    return false;
}
/**
* 注意下面的pollLast和pollFirst方法在链表为空时，会返回null。
*/
public E pollLast() {
    final Node<E> l = last;
    return (l == null) ? null : unlinkLast(l);
}
public E pollFirst() {
    final Node<E> f = first;
    return (f == null) ? null : unlinkFirst(f);
}
```

(4) get()方法

```
public E getFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return f.item;
}
public E getLast() {
```

```
    final Node<E> l = last;
    if (l == null)
        throw new NoSuchElementException();
    return l.item;
}
public E get(int index) {
    checkElementIndex(index);
    return node(index).item;
}
/**
* 根据index的范围判断是从first去寻找还是从last开始去寻找
*/
Node<E> node(int index) {
    // assert isElementIndex(index);

    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
public E element() {
    return getFirst();
}
public E poll() {
    final Node<E> f = first;
    return (f == null) ? null : unlinkFirst(f);
}
public E peek() {
    final Node<E> f = first;
    return (f == null) ? null : f.item;
}
```

(5) 作为队列（offer、poll)

```
/**
* 在队尾添加元素（last后面）
*/
public boolean offer(E e) {
    return add(e);
}
/**
* 移除队首元素（fist）
*/
public E poll() {
    final Node<E> f = first;
    return (f == null) ? null : unlinkFirst(f);
```

```
}
/**
* 获取队首元素，但不删除这个元素，若队列为空，则返回null
*/
public E peek() {
    final Node<E> f = first;
    return (f == null) ? null : f.item;
}
```

（6）作为栈（push、pop）

```
/**
* 进栈
*/
public void push(E e) {
    addFirst(e);
}
/**
* 出栈
*/
public E pop() {
    return removeFirst();
}
public E removeFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return unlinkFirst(f);
}
/**
* 获取栈顶元素
*/
public E peek() {
    final Node<E> f = first;
    return (f == null) ? null : f.item;
}
```

4.其他方法

方法名	用途
void add(int index,Object element)	将元素element插入到list集合的index处，index之后的往后移动
boolean addAll(int index, Collection c)	将集合c所包含的所有元素插入到list集合的index处
Object get(int index)	返回集合index索引处的元素
int indexOf(Object o)	返回对象o在list集合中第一次出现的位置索引
int lastIndexOf(Object o)	返回对象o在list集合中最后一次出现的位置索引
Object remove(int index)	删除并返回index索引处的元素
Object set(int index,Object element)	将index索引处的元素替换成element对象，返回被替换的旧元素
List subList(int fromIndex,int toIndex)	返回从索引fromIndex(包含)到索引toIndex(不包含)处所有集合元素组成的子集
void sort(Comparactor c)	根据c参数对list集合的元素进行排序
void replaceAll(UnaryOperator operator)	根据operation指定的计算规则重新设置List集合的所有元素例如：books.replace(ele->((String)ele).length())使用每个字符串的长度作为新的集合元素。

在接口List中有一个方法：

```
default void sort(Comparator<? super E> c) {
    Object[] a = this.toArray();
    Arrays.sort(a, (Comparator) c);
    ListIterator<E> i = this.listIterator();
    for (Object e : a) {
        i.next();
        i.set((E) e);
    }
}
```

二、Set

比较：

- HashSet的性能总是比TreeSet好，特别是最常用的添加、查询元素等操作。当需要一个保持排序的Set时，才是用TreeSet，否则都使用HashSet。
- LinkedHashSet比HashSet在普通插入、删除操作上要稍微慢一点，但是遍历LinkedHashSet会更快。
- EnumSet是所有Set实现类中性能最好的，但它只保存同一个枚举类的枚举值作为集合元素。

Set的三个实现类HashSet、TreeSet和EnumSet都是线程不安全的。如果有多个线程同时访问一个Set集合，并且有一个线程修改了该Set，则必须手动保证该Set集合的同步性。通常可以通过Collections工具类的synchronizedSortedSet方法来“包装”该Set集合。此操作最好在创建时进行，以防止对Set集合的以外非同步访问。

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(....));
```

1. HashSet

- HashSet内部由HashMap实现，其中key为输入的key，value为new Object()。
- HashSet按hash算法存储集合中的元素，因此具有很好的存取和查找性能。
- HashSet不能保证元素的插入顺序和排列顺序相同。
- HashSet不是线程安全的，必须通过代码来保证其同步。
- 集合元素值可以为null
- 当向HashSet集合中插入一个元素时，HashSet会调用该对象的hashCode()方法来获取hashCode值，然后根据该 hashCode值决定该对象在HashSet中的存放位置。如果有两个元素通过equals()方法比较返回true，但他们的hashCode()方法返回值不相等，HashSet 将会把他们存储在不同位置，依然可以添加成功。也就是说，**HashSet集合判断两个元素相等的标准是两个对象通过equals()方法比较相等并且两个对象的 hashCode()方法返回值也相等。**
- 如果 HashSet 中两个以上的元素具有相同的 hashCode 值，性能将会下降（存在 hashCode 值相同的原因是因为在比较是需要 equals 和 hashCode 方法都相同时才认为是同一个值，否则如果 equals 方法不同但是 hashCode 值相同时，认为不是同一个值，此时在这个 hashCode 值的位置上用链式结构保存多个对象，导致性 能下降。）
- 当程序把可变对象添加到 HashSet 中之后，尽量不要去修改该集合元素中参与计算 hashCode()和 equals()的实例变量，否则将会导致 HashSet无法正确操作这些集合。

（1）构造函数

```
private transient HashMap<E,Object> map;

public HashSet() {
    map = new HashMap<>();
}
/**
* 如果是将一个Collection c作为参数传入，那么HashSet的初始容量为：最接近的2^n的容量
*/
public HashSet(Collection<? extends E> c) {
    map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
    addAll(c);
}
/**
* 指定装在因子的值，表示当数组中有超过loadFactor比例的数据，则对数组进行扩容
*/
public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<>(initialCapacity, loadFactor);
}

很多构造函数...
```

（2）add()方法

```
private static final Object PRESENT = new Object();

public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

(3) remove()方法

```
public boolean remove(Object o) {
    return map.remove(o)==PRESENT;
}
```

2.LinkedHashSet

HashSet 还有一个子类 LinkedHashSet，它也是根据元素的 hashCode值来决定元素的存储位置，但它同时使 用链表维护元素的次序，这样使得元素看起来是以插入的顺序保存的。也就是说，当遍历 LinkedHashSet 集合 里的元素时，LinkedHashSet将会按元素的添加顺序来访问集合里的元素。

LinkedHashSet 需要维护元素的插入顺序，因此性能略低于 HashSet 的性能，但在迭代访问 Set 里的全部元 素时将有很好的性能，因为它以链表来维护内部顺序。

```
LinkedHashSet books = new LinkedHashSet();
books.add("a");
books.add("b");
System.out.println(books);
books.remove("a");
books.add("a");
System.out.println(books);

输出的结果为:
[a,b]
[b,a]
输出 LinkedHashSet集合的元素时，元素的顺序总是与添加顺序一致。
```

三、Queue

Queue用于模拟队列这种数据结构，队列通常是指“先进先出”FIFO的容器。

1.Queue接口中定义的方法

方法	作用
void add(Object o)	将元素e添加到此队列尾部
boolean offer(Object o)	将指定元素加入此队列的尾部，当使用有容量限制的队列时，此方法通常比add(Object o)方法更好
Object element()	获得队列头部元素，但不删除元素
Object peek()	获得队列头部元素，但不删除元素。如果队列为空，则返回null
Object poll()	获取队列头部元素，删除该元素并范湖。如果队列为空，则返回null
Object remove()	获取队列头部元素，并删除该元素

2.PriorityQueue

- PriorityQueue是一个比较标准的队列实现类。
- 它保存元素的顺序是根据元素的大小进行重新排序，而不是加入队列的顺序。因此在调用 poll()和 peek()取出队列中的元素时，并不是取出最先进入队列的元素，而是取出队列中最小的元素。
- 它不允许插入null
- 内部是由Object[]数组实现，采用小顶堆的方式存储。因此存储的时候前size个均为数据，不为空

（1）构造函数

```
// 默认初始数组长度为11
private static final int DEFAULT_INITIAL_CAPACITY = 11;
public PriorityQueue() {
    this(DEFAULT_INITIAL_CAPACITY, null);
}

public PriorityQueue(int initialCapacity) {
    this(initialCapacity, null);
}

public PriorityQueue(int initialCapacity,
                     Comparator<? super E> comparator) {
    // Note: This restriction of at least one is not actually needed,
    // but continues for 1.5 compatibility
    if (initialCapacity < 1)
        throw new IllegalArgumentException();
    this.queue = new Object[initialCapacity];
    this.comparator = comparator;
}
还有一些其他的构造方法
```

（2）offer()方法

```
public boolean offer(E e) {
    // 不允许插入null值
    if (e == null)
        throw new NullPointerException();
```

```
    modCount++;
    int i = size;
    // 当容量不够时，对数组进行扩容
    if (i >= queue.length)
        grow(i + 1);
    size = i + 1;
    if (i == 0)
        queue[0] = e;
    else
        siftUp(i, e);
    return true;
}
/**
* 扩容方式为:
* 当oldCapacity < 64时，扩容为 2*oldCapacity + 2
* 否则，扩容为 2*oldCapacity
*/
private void grow(int minCapacity) {
    int oldCapacity = queue.length;
    // Double size if small; else grow by 50%
    int newCapacity = oldCapacity + ((oldCapacity < 64) ?
                                     (oldCapacity + 2) :
                                     (oldCapacity >> 1));
    // overflow-conscious code
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    queue = Arrays.copyOf(queue, newCapacity);
}
// 真正添加元素的过程
private void siftUp(int k, E x) {
    if (comparator != null)
        siftUpUsingComparator(k, x);
    else
        siftUpComparable(k, x);
}
// PriorityQueue指定了排序方法，使用指定的排序方法进行排序
private void siftUpUsingComparator(int k, E x) {
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = queue[parent];
        if (comparator.compare(x, (E) e) >= 0)
            break;
        // 如果 x < e(parent)，则将parent往下挪动
        queue[k] = e;
        k = parent;
    }
    queue[k] = x;
}
// 根据自己定义的排序工具进行排序
private void siftUpComparable(int k, E x) {
    Comparable<? super E> key = (Comparable<? super E>) x;
    while (k > 0) {
        int parent = (k - 1) >>> 1;
```

```
        Object e = queue[parent];
        if (key.compareTo((E) e) >= 0)
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = key;
}
```

(3) poll()方法

```
public E poll() {
    // 如果队列中没有元素，则返回null
    if (size == 0)
        return null;
    int s = --size;
    modCount++;
    E result = (E) queue[0];
    E x = (E) queue[s];
    queue[s] = null;
    if (s != 0)
        siftDown(0, x);
    return result;
}
// 传入的x参数表示的是堆的 最后一个数字
private void siftDown(int k, E x) {
    if (comparator != null)
        siftDownUsingComparator(k, x);
    else
        siftDownComparable(k, x);
}
// 堆排序中删除元素调整过程
private void siftDownComparable(int k, E x) {
    Comparable<? super E> key = (Comparable<? super E>)x;
    int half = size >>> 1;        // loop while a non-leaf
    while (k < half) {
        int child = (k << 1) + 1; // assume left child is least
        Object c = queue[child];
        int right = child + 1;
        // 获得左右子节点中的最大值
        if (right < size &&
            ((Comparable<? super E>) c).compareTo((E) queue[right]) > 0)
            c = queue[child = right];
        // 如果当前x是最小值就直接退出
        if (key.compareTo((E) c) <= 0)
            break;
        queue[k] = c;
        k = child;
    }
    queue[k] = key;
}

@SuppressWarnings("unchecked")
```

```
private void siftDownUsingComparator(int k, E x) {
    int half = size >>> 1;
    while (k < half) {
        int child = (k << 1) + 1;
        Object c = queue[child];
        int right = child + 1;
        if (right < size &&
            comparator.compare((E) c, (E) queue[right]) > 0)
            c = queue[child = right];
        if (comparator.compare(x, (E) c) <= 0)
            break;
        queue[k] = c;
        k = child;
    }
    queue[k] = x;
}
```

(4) 例子

```
PriorityQueue pq = new PriorityQueue();
pq.offer(6);
pq.offer(-3);
pq.offer(20);
pq.offer(18);
System.out.println(pq);

输出的结果为:
[-3, 6, 20, 18]
```

3.Deque

Deque代表一个“双端队列”，可以同时从两端来添加、删除元素，因此Deque的实现类既可以当做队列使用，也可以当做栈使用。另外参见，LinkedList。

4.ArrayDeque

- ArrayDeque既是List的实现类，也实现了Deque接口，由于实现了Deque接口，因此可以当做栈来使用；
- 而ArrayDeque底层也是基于数组的实现，因此性能也很好。
- 不能插入null
- 采用的是循环队列，队列已满的条件是$tail \mod size = head$。
- 队列是否为空的判断是$tail == head$。

(1) 构造函数

```
transient Object[] elements;
// 表示的是下一个可插入值的位置，即tail位置值为null
transient int tail;
// 表示队列头部位置
transient int head;

public ArrayDeque() {
```

```
    elements = new Object[16];
}

public ArrayDeque(int numElements) {
    allocateElements(numElements);
}

public ArrayDeque(Collection<? extends E> c) {
    allocateElements(c.size());
    addAll(c);
}
```

(2) 作为队列（offer、poll)

```
// 在数组尾部添加元素
public boolean offer(E e) {
    return offerLast(e);
}
public boolean offerLast(E e) {
    addLast(e);
    return true;
}
/**
* 假设当前数组长度为length，在每次完成插入之后，就会检查当前数组是否已经填满，若填满，则扩容为2*length
*/
public void addLast(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[tail] = e;
    if ( (tail = (tail + 1) & (elements.length - 1)) == head)
        doubleCapacity();
}
private void doubleCapacity() {
    assert head == tail;
    int p = head;
    int n = elements.length;
    int r = n - p; // number of elements to the right of p
    int newCapacity = n << 1;
    if (newCapacity < 0)
        throw new IllegalStateException("Sorry, deque too big");
    Object[] a = new Object[newCapacity];
    // 按照顺序拷贝，先拷贝head->数组尾部，再拷贝0->tail
    System.arraycopy(elements, p, a, 0, r);
    System.arraycopy(elements, 0, a, r, p);
    elements = a;
    head = 0;
    tail = n;
}
// 移除队首元素
public E poll() {
    return pollFirst();
}
```

```
public E pollFirst() {
    int h = head;
    @SuppressWarnings("unchecked")
    E result = (E) elements[h];
    // Element is null if deque empty
    if (result == null)
        return null;
    elements[h] = null;     // Must null out slot
    head = (h + 1) & (elements.length - 1);
    return result;
}
```

(3) 作为栈（push、pop）

```
public void push(E e) {
    addFirst(e);
}
public void addFirst(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[head = (head - 1) & (elements.length - 1)] = e;
    if (head == tail)
        doubleCapacity();
}
public E pop() {
    return removeFirst();
}
public E removeFirst() {
    E x = pollFirst();
    if (x == null)
        throw new NoSuchElementException();
    return x;
}
```