时间复杂度为 O(n^2) 的排序算法：

1. 冒泡排序（Bubble Sort）
2. 插入排序（Insertion Sort）
3. 选择排序（Selection Sort）
4. 希尔排序（Shell Sort）

时间复杂度为 O(n log n) 的排序算法：

5. 归并排序（Merge Sort）

1. 快速排序（Quick Sort）
2. 堆排序（Heap Sort）

时间复杂度为线性的排序算法：

8. 计数排序（Counting Sort）

1. 桶排序（Bucket Sort）
2. 基数排序（Radix Sort）

其他排序算法：

11. 梳排序（Comb Sort）- 时间复杂度介于 O(n) 和 O(n^2) 之间

1. 鸽巢排序（Pigeonhole Sort）- 时间复杂度为 O(n + m)，其中 m 是元素的范围
2. 鸡尾酒排序（Cocktail Shaker Sort）- 时间复杂度为 O(n^2)，是冒泡排序的变体
3. 奇偶排序（Odd-Even Sort）- 时间复杂度为 O(n^2)，是冒泡排序的变体

**Comparator**是Java中的一个接口，用于比较两个对象的顺序。它定义了一个**compare()**方法，用于比较两个对象的大小关系。具体来说，**Comparator**接口在集合排序、查找、自定义对象比较等场景中非常有用。

**Comparator**接口定义如下：

public interface Comparator<T> {

int compare(T o1, T o2);

// ...

}

**compare()**方法的返回值有以下规则：

* 如果**o1**小于**o2**，应返回一个负整数。
* 如果**o1**等于**o2**，应返回0。
* 如果**o1**大于**o2**，应返回一个正整数。

根据返回值的不同，可以确定两个对象的相对顺序。这个方法可以根据需求来自定义比较规则，例如按照对象的某个属性进行排序、按照自定义的比较逻辑进行排序等。

在排序操作中，可以通过提供一个实现了**Comparator**接口的比较器对象，来定义对象的排序方式。这样，在排序过程中，算法会使用比较器的**compare()**方法来确定对象的顺序。

比较器的使用示例：

List<Integer> numbers = Arrays.asList(3, 1, 2);

numbers.sort(new Comparator<Integer>() {

public int compare(Integer o1, Integer o2) {

return o1 - o2; // 升序排序

}

});

System.out.println(numbers); // 输出: [1, 2, 3]

在上述示例中，通过传递一个匿名内部类实现的**Comparator**对象给**sort()**方法，定义了整数列表的升序排序规则。**compare()**方法的实现返回**o1 - o2**，即根据数值的大小关系进行比较。

通过使用**Comparator**和实现其**compare()**方法，可以对集合中的对象进行灵活的排序操作，满足不同排序需求。

public class StringLengthComparator implements Comparator<String> {

public int compare(String s1, String s2) {

return Integer.compare(s1.length(), s2.length());

}

}

public class Main {

public static void main(String[] args) {

String[] strings = { "apple", "banana", "cherry", "date" };

StringLengthComparator comparator = new StringLengthComparator();

Arrays.sort(strings, comparator);

for (String str : strings) {

System.out.println(str);

}

}

}

————————————————————————————

**Comparable**是Java中的一个接口，用于定义对象自身的自然排序方式。实现了**Comparable**接口的类可以通过实现**compareTo()**方法来定义对象的比较规则。**Comparable**接口常用于对集合中的对象进行排序。

**Comparable**接口定义如下：

public interface Comparable<T> {

int compareTo(T o);

// ...

}**compareTo()**方法是**Comparable**接口的核心方法，用于比较当前对象与另一个对象的大小关系。它接受一个参数**o**，表示需要比较的另一个对象。**compareTo()**方法返回一个整数值，用于表示当前对象与参数对象的相对顺序。

**compareTo()**方法的返回值有以下规则：

* 如果当前对象小于参数对象，应返回一个负整数。
* 如果当前对象等于参数对象，应返回0。
* 如果当前对象大于参数对象，应返回一个正整数。

根据返回值的不同，可以确定当前对象与参数对象的相对顺序。

实现了**Comparable**接口的类可以直接使用Java提供的排序方法（如**Collections.sort()**或**Arrays.sort()**）进行排序，而无需额外提供比较器对象。

使用**Comparable**接口的示例：

public class Person implements Comparable<Person> {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// 实现compareTo()方法

public int compareTo(Person other) {

return this.age - other.age; // 按照年龄进行排序

}

}在上述示例中，**Person**类实现了**Comparable<Person>**接口，并实现了**compareTo()**方法来定义人员对象的比较规则。在这个例子中，根据人员的年龄来进行比较，返回**this.age - other.age**来表示相对顺序。

通过实现**Comparable**接口和**compareTo()**方法，对象可以定义自身的比较规则，并能够直接使用Java提供的排序方法进行排序操作，非常方便和简洁。

public class Number implements Comparable<Number> {

private int value;

public Number(int value) {

this.value = value;

}

public int getValue() {

return value;

}

public int compareTo(Number other) {

return Integer.compare(this.value, other.value);

}

public static void main(String[] args) {

Number num1 = new Number(10);

Number num2 = new Number(5);

int result = num1.compareTo(num2);

if (result < 0) {

System.out.println("num1 is smaller than num2");

} else if (result > 0) {

System.out.println("num1 is greater than num2");

} else {

System.out.println("num1 is equal to num2");

}

}

}

**Selection Sort**

public void selectionSort(E[] data, int size, Comparator<E> comp) {

// 针对每个位置，从0开始，找到下一个最小的元素并将其交换到正确的位置上

for (int place = 0; place < size - 1; place++) {

int minIndex = place; // 记录当前最小元素的索引

for (int sweep = place + 1; sweep < size; sweep++) {

// 使用比较器 comp 比较元素，找到更小的元素

if (comp.compare(data[sweep], data[minIndex]) < 0)

minIndex = sweep;

}

// 将当前最小元素与当前位置的元素进行交换

swap(data, place, minIndex);

}

}

**Merge Sort**

public class MergeSortZXY {

    public static void myMergeSort(int[] arr, int left, int right) {

        if (left >= right) {

            return;

        }

        int mid = (left + right) / 2;

        myMergeSort(arr, left, mid);// 左半部分 left --- mid

        myMergeSort(arr, mid + 1, right);// 右半部分 mid+1 --- right

        myMerge(arr, left, mid, right);// 最后合并

    }

    public static void myMerge(int[] arr, int left, int mid, int right) {

        // 创建一个临时数组，用于存放合并后的结果

        int[] temp = new int[right - left + 1];

        // 定义两个指针，分别指向两个子数组的起始位置

        int i = left;

        int j = mid + 1;

        // 定义一个索引，用于记录临时数组中元素的位置

        int k = 0;

        while (i <= mid && j <= right) { // 合并数据到临时数组temp

            if (arr[i] <= arr[j]) {

                temp[k++] = arr[i++];

            } else {

                temp[k++] = arr[j++];

            }

        }

        // 当左边的指针还没有越界时，将剩余的元素放入临时数组中

        while (i <= mid) {

            temp[k++] = arr[i++];

        }

        // 当右边的指针还没有越界时，将剩余的元素放入临时数组中

        while (j <= right) {

            temp[k++] = arr[j++];

        }

        // 将临时数组中的元素复制回原始数组中

        for (int p = 0; p < temp.length; p++) {

            arr[left + p] = temp[p];

        }

    }

**Quick Sort**

public class QuickSort {

    public static void quickSort(int[] arr) {

        quickSort(arr, 0, arr.length - 1);

    }

    private static void quickSort(int[] arr, int low, int high) {

        if (low < high) {

            int partitionIndex = partition(arr, low, high);

            quickSort(arr, low, partitionIndex - 1);

            quickSort(arr, partitionIndex + 1, high);

        }

    }

    private static int partition(int[] arr, int low, int high) {

        int pivot = arr[high];

        int i = low - 1;

        for (int j = low; j < high; j++) {

            if (arr[j] < pivot) {

                i++;

                swap(arr, i, j);

            }

        }

        swap(arr, i + 1, high);

        return i + 1;

    }

    private static void swap(int[] arr, int i, int j) {

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }

**前序：左打点(根左右)**

**中序：下打点(左根右)**

**后序：右打点(左右根)**

**BST若删除节点左右都有孩子，则把左子树最大值/右子树最小值复制到该节点上。**

**AVl tree操作均为O(logn)**

**树的ADT:**

public boolean isLeaf() {

        return (right == null) && (left == null);

    }

public BinaryTree<V> find(V val) {

        if (value.equals(val))

            return this;

        if (left != null) {

            BinaryTree<V> ans = left.find(val);

            if (ans != null)

                return ans;

        }

        if (right != null) {

            BinaryTree<V> ans = right.find(val);

            if (ans != null)

                return ans;

        }

        return null;

    }

**前序遍历二叉树：**

public static void printAll(BinaryTree<String> tree, String indent) {

        System.out.println(indent + tree.getValue());

        if (tree.getLeft() != null)

            printAll(tree.getLeft(), indent + " ");

        if (tree.getRight() != null)

            printAll(tree.getRight(), indent + " ");

}或者

System.out.println(indent + tree.getValue());

        for (GeneralTree<String> child : tree.getChildren())

            printAll(child, indent + " ");

    }

处理**哈希冲突**的方法，用于解决多个键映射到相同哈希桶的情况。

1. 开放寻址法（Open Addressing）- 冲突解决方法
   * 使用数组来存储键值对。
   * 线性探测（Linear Probing）：当发生冲突时，通过一定的步长（通常为1）逐个检查下一个槽位，直到找到一个空槽位为止。
   * 二次探测（Quadratic Probing）：根据一定的步长增量（通常为常数倍数或二次方倍数），逐渐检查下一个槽位，以寻找空槽位。
   * 双重哈希（Double Hashing）：使用两个哈希函数，根据一定的步长增量，逐渐检查下一个槽位，直到找到一个空槽位为止。
2. 分离链接法（Separate Chaining）
   * 每个哈希桶使用一个链表（SearchList）来存储具有相同哈希值的键值对。
   * 数组中的每个槽位都可以存储多个键值对，因此不会出现槽位满的情况。
   * 删除操作不会影响其他键值对。

开放寻址法和分离链接法是处理哈希冲突的两种常见方法，它们在解决冲突的方式上有所不同。

* 开放寻址法通过在哈希表中探测下一个可用的槽位来解决冲突，当找到一个空槽位时，将键值对存储在该槽位中。线性探测、二次探测和双重哈希是开放寻址法的具体实现方法，它们使用不同的探测方式来查找下一个空槽位。然而，开放寻址法可能会导致主要聚集（Primary Clustering）和次要聚集（Secondary Clustering）问题。
* 分离链接法通过在每个槽位上使用链表来存储具有相同哈希值的键值对。当发生冲突时，将新的键值对添加到对应槽位的链表中。由于每个槽位都可以存储多个键值对，所以不会出现槽位满的情况。删除操作也相对简单，只需要在链表中删除相应的节点即可。

选择适当的冲突解决方法取决于具体的应用场景和性能需求。开放寻址法在空间效率方面更好，但可能导致聚集问题。而分离链接法可以避免聚集问题，但需要额外的空间来存储链表节点。

**度序列是指将图中所有顶点的度按非递增顺序排列的序列，如果有重复的度，则重复出现。**

**在一个简单图中，所有顶点的度之和等于边的数量的两倍。这是因为每条边连接了两个顶点，因此每个顶点的度都会对应计算两次，所以所有顶点的度之和必然等于边的数量的两倍。**

假设有一个简单图，其中顶点A的度为3，顶点B的度为2，顶点C的度为4。那么度序列就是4、3、2。而边的数量为(3+2+4)/2 = 4，符合性质中的关系。这个性质在图论中非常有用，可以用来验证图的正确性或计算图的性质。

* 一条由形式为VsVi、ViVj、VjVk、VkVl、VlVt的边组成的序列是从顶点Vs到Vt的一条路径（walk）。
* 如果这些边是不同的，那么这条路径被称为轨迹（trail）。
* 如果路径中的顶点也是不同的，那么这条路径被称为路径（path）。
* 如果路径的起始点和终点相同，即Vs = Vt，那么这条路径被称为闭合路径（closed walk）。
* 如果闭合路径中除了起始点和终点之外的所有顶点都是不同的，那么这条路径被称为循环（cycle）或电路（circuit）。
* 一条路径中边的数量被称为路径的长度

Trail（轨迹）：Trail是一条由边组成的序列，其中边可以重复经过同一个顶点，但顶点必须按顺序连接。也就是说，Trail中的边是不同的，但顶点可以重复。Path（路径）：Path是一条由边组成的序列，其中既不重复经过边，也不重复经过顶点。路径中的边和顶点都是唯一的。因此，可以通过观察序列中的顶点是否重复来区分Trail和Path。如果顶点可以重复出现，则为Trail；如果顶点不重复出现，则为Path。

**TTL**

**Bag中Add**自动扩容方法：

public void add(int element) {

      if (manyItems == data.length) {

         ensureCapacity((manyItems + 1) \* 2); // Ensure twice as much space as we need.

      }

      data[manyItems] = element;

      manyItems++;

   }

public void addMany(int... elements) {

      if (manyItems + elements.length > data.length) {

         ensureCapacity((manyItems + elements.length) \* 2);// Ensure twice as much space as we need.

      }

      System.arraycopy(elements, 0, data, manyItems, elements.length);

      manyItems += elements.length;

   }

**Change the current capacity of this bag:**

   public void ensureCapacity(int minimumCapacity) {

      int[] biggerArray;

      if (data.length < minimumCapacity) {

         biggerArray = new int[minimumCapacity];

         System.arraycopy(data, 0, biggerArray, 0, manyItems);

         data = biggerArray;

      }

   }

**与ArrayList不同，数组存储的是实际的元素值，而不是对象的引用。每个数组元素都直接存储了其对应类型的值，而不是引用其他内存位置上的对象。因此，数组中的元素在内存中是连续存储的。**

**虽然 String 是 Object 的子类，但泛型类型之间并不具有继承关系。List<String> 并不直接继承自 List<Object>。**

**泛型**

public void myMethod(Integer i)：这是一个普通的方法，接受一个 Integer 类型的参数。

public void myMethod(Double d)：这也是一个普通的方法，接受一个 Double 类型的参数。

然后，我们还有一个泛型方法的声明： public <T> void myMethod(T t)：这是一个泛型方法，其中 <T> 表示类型参数。这个方法可以接受任意类型的参数，并在方法内部使用类型参数 T 进行操作。

**Bounded Type Parameters：**

**二叉树类型：**

1.满二叉树（Full binary tree）： 满二叉树是一种特殊的二叉树，其中每个节点要么没有子节点（叶节点），要么有两个子节点。换句话说，每个节点的度要么为0，要么为2。满二叉树的一个特点是，所有的叶节点都在同一层上。满二叉树的高度可以通过节点数量计算得出：假设树的高度为h，则节点数量为2^h - 1。

2.完美二叉树（Perfect binary tree）： 完美二叉树是一种特殊的二叉树，所有的内部节点都有两个子节点，并且所有的叶节点都在同一层上。换句话说，每个节点的度都为2，且所有的叶节点都在同一层上。完美二叉树的高度是固定的，可以通过内部节点的数量计算得出：假设内部节点数量为n，则树的高度为log2(n+1)。

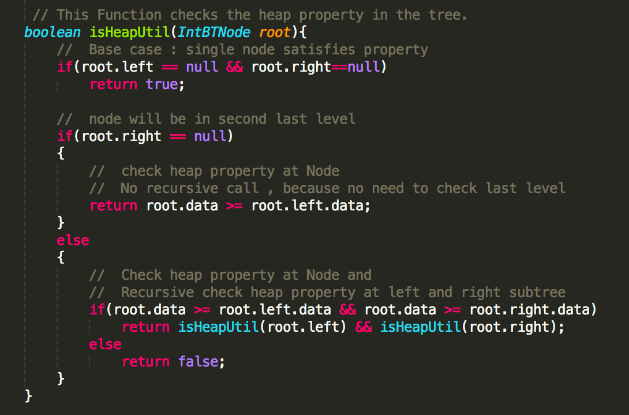
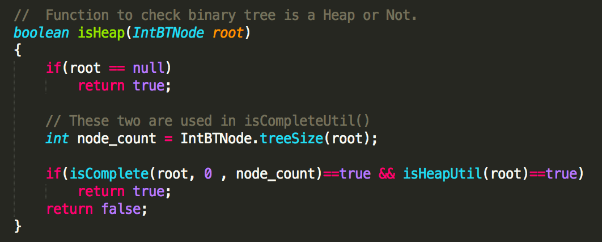
3.完全二叉树（Complete binary tree）： 完全二叉树是一种二叉树，除了最后一层可能不满外，其他每一层都是完全填充的，并且所有的节点都尽可能地靠左排列。换句话说，从根节点到倒数第二层的节点都是满的，最后一层的节点靠左排列。完全二叉树在实际应用中比较常见。完全二叉树的高度取决于节点数量，但不一定是固定的。

\*当我们将完全二叉树按照从上到下、从左到右的顺序转换为数组时，可以利用以下规律来确定节点在数组中的位置关系：

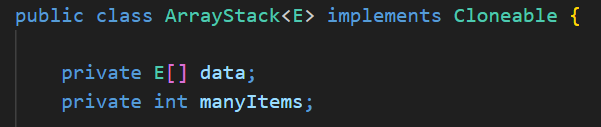
1. 根节点位于数组的索引 [0]。
2. 对于任意节点在数组中的索引为 [i]，其父节点的索引可以通过计算得出：父节点索引为 ⌊(i-1)/2⌋（其中 ⌊ ⌋ 表示向下取整），但要排除根节点的情况（i≠0）。
3. 左子节点位于索引 [2i+1]。
4. 右子节点位于索引 [2i+2]。

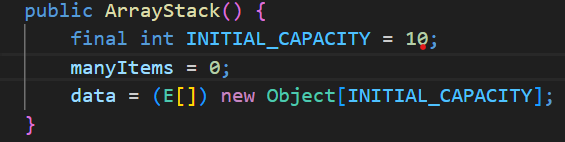
这样的表示方法可以有效地利用数组的连续存储特性来表示完全二叉树的结构，无需使用额外的指针或引用。通过简单的索引计算，我们可以在数组中快速访问和操作完全二叉树的节点。

**判断一棵树是否为完全二叉树的方法：**

1. 首先，计算二叉树中节点的总数（count）。
2. 从根节点开始递归遍历二叉树。在遍历过程中，我们维护一个索引（i），初始值为0，并将节点总数（count）传入递归函数。
3. 如果当前遍历到的节点为空（NULL），则说明该节点是一个空节点，表示树的一部分已经结束。在完全二叉树中，空节点只会出现在最后一层，因此如果遇到空节点，我们可以判断该树是完全二叉树。返回true。
4. 如果当前节点的索引（i）大于或等于节点总数（count），则说明索引超出了树中实际存在的节点范围，这意味着树不是完全二叉树。返回false。
5. 接下来，我们递归检查左子树和右子树是否满足完全二叉树的条件。对于左子树，我们更新索引为（2*i+1）；对于右子树，我们更新索引为（2*i+2）。
6. 如果左子树和右子树都满足完全二叉树的条件，则整棵树是完全二叉树。返回true；否则，返回false。
7. 
8. 

**ArrayStack:**

****

****

public E pop() {

        if (manyItems == 0)

            // EmptyStackException is from java.util and its constructor has no

            // argument.

            throw new EmptyStackException();

        return data[--manyItems];

    }

public void push(E item) {

        if (manyItems == data.length) {

            // Double the capacity and add 1; this works even if manyItems is 0.

            // However, in

            // case that manyItems\*2 + 1 is beyond Integer.MAX\_VALUE, there will

            // be an

            // arithmetic overflow and the bag will fail.

            ensureCapacity(manyItems \* 2 + 1);

        }

        data[manyItems] = item;

        manyItems++;

    }

**完整版ArrayStack：**

import java.util.EmptyStackException;

public class ArrayStack<T> {

private static final int DEFAULT\_CAPACITY = 10;

private T[] array;

private int size;

public ArrayStack() {

array = (T[]) new Object[DEFAULT\_CAPACITY];

size = 0;

}

public ArrayStack(int capacity) {

if (capacity < 0) {

throw new IllegalArgumentException("Capacity cannot be negative.");

}

array = (T[]) new Object[capacity];

size = 0;

}

public void push(T element) {

if (size == array.length) {

resizeArray(array.length \* 2);

}

array[size] = element;

size++;

}

public T pop() {

if (isEmpty()) {

throw new EmptyStackException();

}

T element = array[size - 1];

array[size - 1] = null;

size--;

if (size > 0 && size == array.length / 4) {

resizeArray(array.length / 2);

}

return element;

}

public T peek() {

if (isEmpty()) {

throw new EmptyStackException();

}

return array[size - 1];

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

private void resizeArray(int capacity) {

T[] newArray = (T[]) new Object[capacity];

System.arraycopy(array, 0, newArray, 0, size);

array = newArray;

}

}

**后缀表达式**

换为后缀表达式的过程如下：

1. 创建一个空堆栈和一个空的输出队列。
2. 从左到右扫描表达式的每个字符：

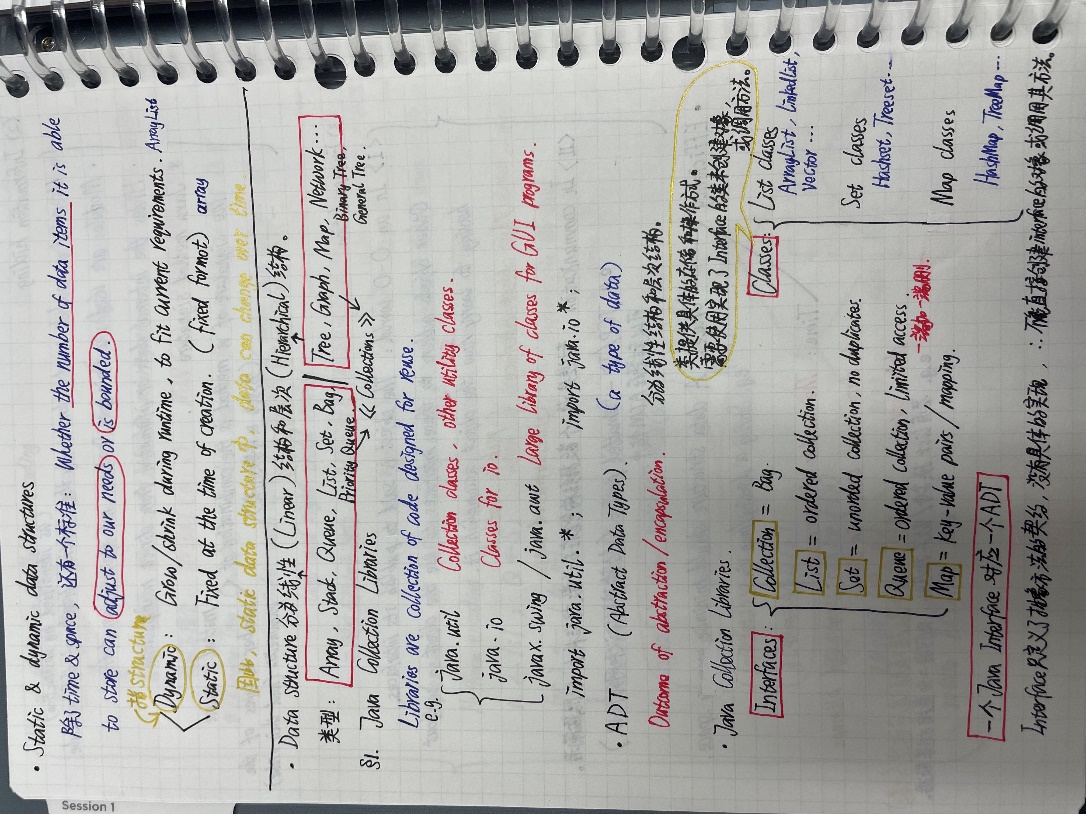
当遇到操作数（数字）时，直接将其添加到输出队列中。

当遇到运算符时，根据优先级判断是否需要将其推入堆栈：

当遇到左括号"("时，将其推入堆栈中。

当遇到右括号")"时，将堆栈中的运算符依次弹出并添加到输出队列中，直到遇到左括号为止。左括号不添加到输出队列，而是丢弃。

1. 扫描完表达式后，将堆栈中剩余的运算符依次弹出并添加到输出队列中。
2. 输出队列中的表达式即为转换后的后缀表达式。



private void ensureCapacity(int minCapacity) {

int oldCapacity = elementData.length;

if (minCapacity > oldCapacity) {

int newCapacity = (oldCapacity \* 3) / 2 + 1;

if (newCapacity < minCapacity)

newCapacity = minCapacity;

elementData = Arrays.copyOf(elementData, newCapacity);

}

}

