1. 排序算法

我们之前的学习已经对所有排序方法都有了了解，这里稍微复习一下。

1.1 冒泡排序（Bubble sort）

冒泡排序是一种简单的排序算法，它重复地遍历待排序的列表，比较每一对相邻的元素，如果它们的顺序错误就交换它们。每次遍历都会让最大的元素"冒泡"到列表的末尾。这个过程会重复进行，直到整个列表有序。
代码如下。

```
public class BubbleSort {
    public static void bubbleSort(int[] list) {
        for (int k = 1; k < list.length; k++) { // 外层循环控制趟数
            for (int i = 0; i < list.length - k; i++) { // 内层循环比较相邻元素
                if (list[i] > list[i + 1]) { // 如果当前元素大于后面的元素
                    // 交换它们
                    int temp = list[i];
                    list[i] = list[i + 1];
                    list[i + 1] = temp;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] list = {64, 34, 25, 12, 22, 11, 90};
        bubbleSort(list); // 调用冒泡排序
        System.out.println("Sorted list:");
        for (int num : list) {
            System.out.print(num + " ");
        }
    }
```

```
}
```

如果列表几乎已经排序好了，或者已经完全排序好了，传统的冒泡排序算法仍然会进行不必要的遍历和比较，这会浪费计算资源。
下面的代码会引入一个布尔变量 needNextPass 来检测数组是否可能已经排序好了，从而避免不必要的遍历。

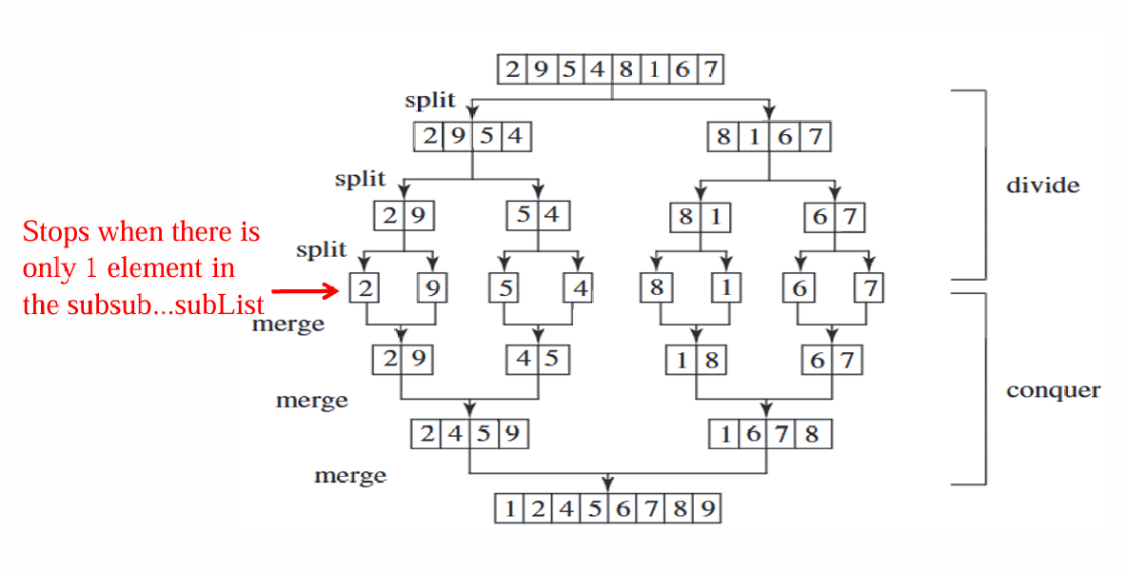
```java
public class BubbleSort {
    public static void bubbleSort(int[] list) {
        boolean needNextPass = true;
        for (int k = 1; k < list.length && needNextPass; k++) {
            needNextPass = false; // 假设这一轮不需要再排序
            for (int i = 0; i < list.length - k; i++) {
                if (list[i] > list[i + 1]) {
                    int temp = list[i];
                    list[i] = list[i + 1];
                    list[i + 1] = temp;
                    needNextPass = true; // 发生了交换，说明还需要继续排序
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] list = {1, 2, 3, 4};
        bubbleSort(list);
        System.out.println("Sorted list:");
        for (int num : list) {
            System.out.print(num + " ");
        }
    }
}
```

冒泡排序的时间复杂度再最佳情况下是$O(n)$，在最坏情况下（数组完全逆序）是$O(n^2)$。

1.2 归并排序（Merge Sort）

归并排序使用分治法（Divide and Conquer）策略，因此详细步骤如下：

1. 分解（Divide）：
   将数组分成两半。如果数组的长度是奇数，那么可以将中间的元素单独处理，或者将其归入左边或右边的子数组。
   这个过程是递归进行的，即对每个子数组继续进行分解，直到每个子数组只包含一个元素。由于只有一个元素的数组自然是有序的，所以这是递归的终止条件。
2. 解决（Conquer）：
   对每个子数组进行排序。由于子数组是由原始数组分解而来，且每个子数组只包含一个元素，所以这一步实际上是在递归地对子数组进行排序。
3. 合并（Combine）：
   将排序好的子数组合并成一个有序数组。这一步是归并排序的核心，需要将两个已排序的子数组合并成一个有序数组。
   合并过程是通过比较两个子数组的首元素，将较小的元素放入新数组中，然后从相应的子数组中移除该元素，重复这个过程直到两个子数组中的元素都被合并到新数组中。



代码如下。

```
public class MergeSortTest {
    public static void mergeSort(int[] list) {
        if (list.length > 1) {
            // 分解第一个子数组
            int[] firstHalf = new int[list.length / 2];
            System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
            mergeSort(firstHalf); // 递归排序第一个子数组

            // 分解第二个子数组
            int secondHalfLength = list.length - list.length / 2;
            int[] secondHalf = new int[secondHalfLength];
            System.arraycopy(list, list.length / 2, secondHalf, 0,
secondHalfLength);
            mergeSort(secondHalf); // 递归排序第二个子数组

            // 合并两个子数组到原数组
            merge(firstHalf, secondHalf, list);
        }
    }

    public static void merge(int[] list1, int[] list2, int[] list) {
        int current1 = 0; // 当前索引在 list1 中
        int current2 = 0; // 当前索引在 list2 中
        int current3 = 0; // 当前索引在 list 中

        // 合并两个子数组
        while (current1 < list1.length && current2 < list2.length) {
            if (list1[current1] < list2[current2]) {
                list[current3++] = list1[current1++];
            } else {
                list[current3++] = list2[current2++];
            }
        }

        // 复制剩余的元素（如果有的话）
        while (current1 < list1.length) {
            list[current3++] = list1[current1++];
        }
```

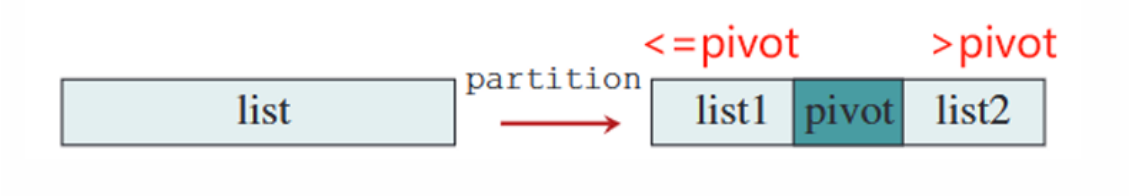
```
        while (current2 < list2.length) {
            list[current3++] = list2[current2++];
        }
    }

    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        long startTime = System.currentTimeMillis();
        mergeSort(a);
        long endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + "ms");
    }

    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++) {
            a[i] = (int) (Math.random() * a.length);
        }
    }
}
```

归并排序的递归关系式为：
$T(n) = T(n/2) + T(n/2) + 2n - 1$
其中：
第一个$T(n/2)$表示对数组的前半部分进行排序所需的时间。
第二个$T(n/2)$ 表示对数组的后半部分进行排序所需的时间。
$2n - 1$表示合并两个已排序子数组所需的时间，包括$n - 1$次比较（用于比较两个子数组的元素）和n次移动（将每个元素放入临时数组）。
因此归并排序的时间复杂度为$O(nlogn)$。

1.3 快速排序（Quick Sort）

快速排序也是使用分治法（Divide and Conquer）策略来实现排序。

1. 算法从数组中选择一个元素，称为基准（pivot）。
2. 将数组分成两部分：
   第一部分（list1）包含所有小于或等于基准的元素。
   第二部分（list2）包含所有大于基准的元素。
   这个过程称为分区（partitioning）。
3. 递归排序：
   对第一部分（list1）和第二部分（list2）分别递归地应用快速排序算法，直到每个子数组只包含一个元素或为空。



代码如下。

```
public class QuickSortTest {
    public static void quickSort(int[] list) {
        quickSort(list, 0, list.length - 1);
```

```
    }

    public static void quickSort(int[] list, int first, int last) {
        if (last > first) {
            int pivotIndex = partition(list, first, last);
            quickSort(list, first, pivotIndex - 1);
            quickSort(list, pivotIndex + 1, last);
        }
    }

    public static int partition(int[] list, int first, int last) {
        int pivot = list[first]; // Choose the first element as pivot
        int low = first + 1; // Index for forward search
        int high = last; // Index for backward search

        while (high > low) {
            // Search forward from left
            while (low <= high && list[low] <= pivot)
                low++;
            // Search backward from right
            while (low <= high && list[high] > pivot)
                high--;
            // Swap two elements in the list
            if (high > low) {
                int temp = list[high];
                list[high] = list[low];
                list[low] = temp;
            }
        }
        // Account for duplicated elements:
        while (high > first && list[high] >= pivot)
            high--;
        // Swap pivot with list[high]
        if (pivot > list[high]) {
            list[first] = list[high];
            list[high] = pivot;
            return high;
        } else {
            return first;
        }
    }

    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        long startTime = System.currentTimeMillis();
        quickSort(a);
        long endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + "ms");
    }

    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
```

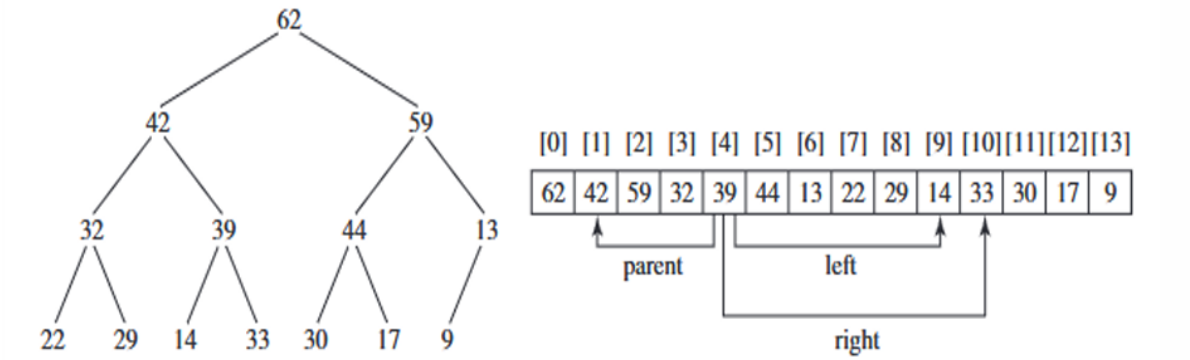
```
    }
}
```

快速排序的递归关系式为：
$T(n) = T(n/2) + T(n/2) + n$
其中：
第一个$T(n/2)$表示对数组的前半部分进行排序所需的时间。
第二个$T(n/2)$ 表示对数组的后半部分进行排序所需的时间。
n表示分区所需的时间。
因此快速排序的时间复杂度为$O(nlogn)$。

然而在最坏情况下，每次分区都将数组分成一个非常大的子数组和一个空的子数组。如，假设数组的第一个元素是分区的基准：
$[1, 2, 3, 4, 5, \ldots, n], size = n$
第一次分区，基准为1：
左侧：空，右侧：$[2, 3, 4, 5, \ldots, n]$，右侧子数组大小为$n - 1$。
第二次分区，基准为2：
左侧：空，右侧：$[3, 4, 5, 6, \ldots, n]$，右侧子数组大小为$n - 2$。
以此类推，直到子数组大小为1。
由于每次分区都将数组分成一个非常大的子数组和一个空的子数组，因此需要递归地对$n - 1$个子数组进行分区操作。
每个分区操作的时间复杂度为$O(n)$，因此总的时间复杂度为$O(n^2)$。

1.4 堆排序（Heap Sort）

这个算法在本学期的算法课上详细介绍过。可以看这里的文章。
相关文章
在堆排序算法中，通常使用完全二叉树（Complete Binary Tree）来实现堆结构。完全二叉树是一种特殊的二叉树，其中除了最后一层外，每一层都是满的，并且最后一层的节点尽可能地集中在左侧。
堆排序算法通过维护一个堆结构来实现排序，堆结构有两种类型：
最大堆（Max Heap）：每个节点的值都大于或等于其子节点的值。
最小堆（Min Heap）：每个节点的值都小于或等于其子节点的值。
堆可以存储在 ArrayList 或数组中，前提是堆的大小是已知的。
对于数组中位置为i的节点，其左子节点的位置是$2i + 1$，右子节点的位置是$2i + 2$，父节点的位置是$(i - 1)/2$（注意使用整数除法）。
以位置为4的节点为例，其两个子节点的位置分别是$2 * 4 + 1 = 9$和$2 * 4 + 2 = 10$，其父节点的位置是（$4 - 1$）$/2 = 1$（注意这里使用的是整数除法，结果为1，而不是1.5）。

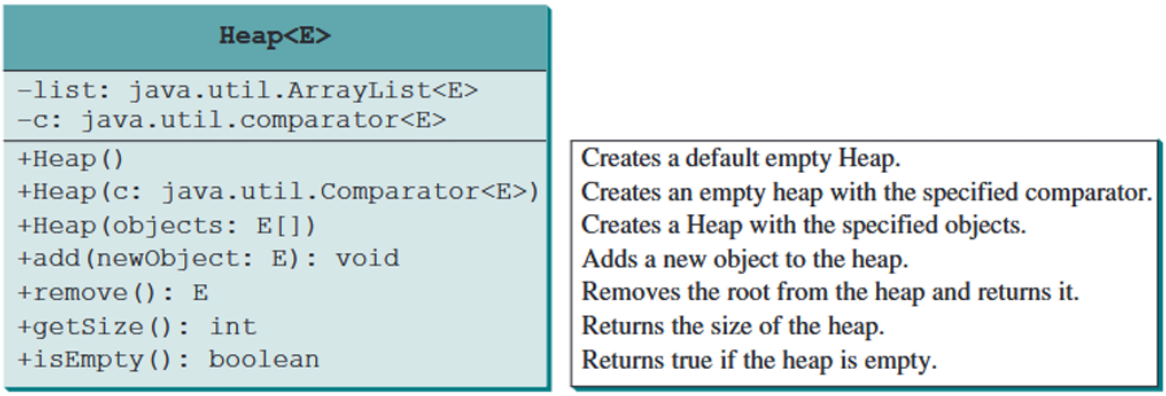


在堆排序中，通常使用最大堆来实现。算法的基本步骤如下：

1. 将待排序的数组转换为最大堆。
2. 将堆顶元素（最大值）与数组的最后一个元素交换，然后将数组缩小一个元素。
3. 重新调整堆，使其满足最大堆的性质。

4. 重复步骤 2 和 3，直到数组完全有序。

因此实现这个算法需要依赖于堆（Heap）这种数据结构。
其UML图如下。



代码如下。

```
import java.util.ArrayList;

public class Heap<E extends Comparable> {
    private ArrayList<E> list = new ArrayList<E>();

    // Create a default heap
    public Heap() {
    }

    // Create a heap from an array of objects
    public Heap(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }

    // Add a new object into the heap
    public void add(E newObject) {
        list.add(newObject); // Append to the end of the heap
        int currentIndex = list.size() - 1; // The index of the last node
        while (currentIndex > 0) {
            int parentIndex = (currentIndex - 1) / 2;
            // Swap if the current object is greater than its parent
            if (list.get(currentIndex).compareTo(list.get(parentIndex)) > 0) {
                E temp = list.get(currentIndex);
                list.set(currentIndex, list.get(parentIndex));
                list.set(parentIndex, temp);
            } else
                break; // the tree is a heap now
            currentIndex = parentIndex;
        }
    }

    // Remove the root from the heap
    public E remove() {
        if (list.size() == 0) return null;
        E removedObject = list.get(0);
        list.set(0, list.get(list.size() - 1));
        list.remove(list.size() - 1);
```

```
        int currentIndex = 0;
        while (currentIndex < list.size()) {
            int leftChildIndex = 2 * currentIndex + 1;
            int rightChildIndex = 2 * currentIndex + 2;
            // Find the maximum between two children
            if (leftChildIndex >= list.size())
                break; // The tree is a heap
            int maxIndex = leftChildIndex;
            if (rightChildIndex < list.size())
                if (list.get(maxIndex).compareTo(list.get(rightChildIndex)) < 0)
                    maxIndex = rightChildIndex;

            // Swap if the current node is less than the maximum
            if (list.get(currentIndex).compareTo(list.get(maxIndex)) < 0) {
                E temp = list.get(maxIndex);
                list.set(maxIndex, list.get(currentIndex));
                list.set(currentIndex, temp);
                currentIndex = maxIndex;
            } else
                break; // The tree is a heap
        }
        return removedObject;
    }

    // Get the number of nodes in the tree
    public int getSize() {
        return list.size();
    }
}
```

所以堆排序的完整代码如下。

```
import java.util.ArrayList;

public class HeapSort {
    public static <E extends Comparable> void heapSort(E[] list) {
        // Create a Heap of E
        Heap<E> heap = new Heap<E>();

        // Add elements to the heap
        for (int i = 0; i < list.length; i++)
            heap.add(list[i]);

        // Remove the highest elements from the heap
        // and store them in the list from end to start
        for (int i = list.length - 1; i >= 0; i--)
            list[i] = heap.remove();
    }

    /** A test method */
    public static void main(String[] args) {
        Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
        heapSort(list);
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
```

```
    }
}

class Heap<E extends Comparable> {
    private ArrayList<E> list = new ArrayList<E>();

    // Create a default heap
    public Heap() {
    }

    // Create a heap from an array of objects
    public Heap(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }

    // Add a new object into the heap
    public void add(E newObject) {
        list.add(newObject); // Append to the end of the heap
        int currentIndex = list.size() - 1; // The index of the last node
        while (currentIndex > 0) {
            int parentIndex = (currentIndex - 1) / 2;
            // Swap if the current object is greater than its parent
            if (list.get(currentIndex).compareTo(list.get(parentIndex)) > 0) {
                E temp = list.get(currentIndex);
                list.set(currentIndex, list.get(parentIndex));
                list.set(parentIndex, temp);
            } else
                break; // the tree is a heap now
            currentIndex = parentIndex;
        }
    }

    // Remove the root from the heap
    public E remove() {
        if (list.size() == 0) return null;
        E removedObject = list.get(0);
        list.set(0, list.get(list.size() - 1));
        list.remove(list.size() - 1);
        int currentIndex = 0;
        while (currentIndex < list.size()) {
            int leftChildIndex = 2 * currentIndex + 1;
            int rightChildIndex = 2 * currentIndex + 2;
            // Find the maximum between two children
            if (leftChildIndex >= list.size())
                break; // The tree is a heap
            int maxIndex = leftChildIndex;
            if (rightChildIndex < list.size())
                if (list.get(maxIndex).compareTo(list.get(rightChildIndex)) < 0)
                    maxIndex = rightChildIndex;

            // Swap if the current node is less than the maximum
            if (list.get(currentIndex).compareTo(list.get(maxIndex)) < 0) {
                E temp = list.get(maxIndex);
                list.set(maxIndex, list.get(currentIndex));
```

```
                list.set(currentIndex, temp);
                currentIndex = maxIndex;
            } else
                break; // The tree is a heap
        }
        return removedObject;
    }

    // Get the number of nodes in the tree
    public int getSize() {
        return list.size();
    }
}
```

堆排序算法的时间复杂度为$O(nlogn)$。
需要一个临时数组来合并两个子数组。这个临时数组的大小与原数组相同，因此归并排序的空间复杂度是 O(n)，其中 n 是数组的长度。
而堆排序不需要额外的数组空间，因此在空间效率上优于归并排序。

2. 在面向对象编程中终身学习

终身学习：指的是持续的、自愿的、自我激励的知识追求，目的是为了个人或职业发展。
在 OOP 领域的应用：终身学习鼓励开发者跟上不断发展的概念、技术和工具，以提高他们在设计、实现和维护软件系统方面的熟练程度。

终身学习的重要性如下：

1. 不断演变的技术环境：
   终身学习确保开发者能够跟上最新的最佳实践、框架和设计模式。
2. 提高专业能力：
   终身学习增强了你解决复杂问题、优化代码和设计健壮系统的能力。
3. 适应新工具：
   学习 OOP 中的新工具使你保持敏捷，并为你打开新的职业机会。
4. 个人成长：
   不断提高你的 OOP 技能不仅增强了你的技术专长，还发展了其他宝贵的技能，如批判性思维、解决问题和有效沟通。

2.1 记录和反思学习过程

我们可以记录和反思学习过程。

1. 维护反思日志/记录：
   记录每个学习阶段的关键收获。
   记录在解决编程挑战后获得的见解和思考。
   记录需要进一步澄清的问题或领域。
   记录你已经工作过的代码片段和示例。
   日志有助于巩固你的学习，并为未来的问题提供参考。
2. 定期自我评估：
   定期评估你与设定目标的进展。你是否掌握了基础知识，或者需要重新审视某些概念。
   诚实地面对挑战。你是否在复杂的设计模式或高级 OOP 原则的实现上遇到困难？接下来专注于这些领域。
   认可你的成就。完成课程或参与 OOP 项目是一个值得庆祝的重要里程碑。

3. 加入社区和论坛：
   加入像 StackOverflow、Reddit 、GitHub 或 CSDN 这样的专注于 OOP 的社区。
   参与讨论，提问，并与他人分享你的知识。

2.2 EDI 原则

1. 平等（Equality）：
   在 OOP 中，平等意味着所有开发者，无论性别、种族、性取向或残疾状况，都有平等的职业机会和参与软件开发过程的能力。
2. 多样性（Diversity）：
   在 OOP 中，多样性通过汇集来自不同背景、经验和观点的人来丰富问题解决。这种多样性导致更创造性和创新性的解决方案来应对复杂的软件设计挑战。
3. 包容性（Inclusion）：
   在 OOP 中，包容性超越了代表性和平等；它强调创造一个每个人都感到尊重和有价值的环境。

将 EDI 融入 OOP 实践便是：

1. 采用包容性编码实践：
   在代码、注释和文档中使用包容性语言。避免可能无意中疏远或排除个体的术语或实践。
2. 实施可访问的设计模式：
   确保你构建的软件设计对广泛受众可访问。纳入可访问性功能，如可定制的 UI 选项、语音识别和高对比度主题。
3. 鼓励跨多样化团队的协作：
   积极寻求来自不同背景和经验的团队成员。这可能意味着促进科技领域的女性、支持 LGBTQ+ 个体，或与包括来自不同种族和文化的人的团队合作。
4. 利用 EDI 聚焦资源：
   利用专注于软件开发社区多样性和包容性的资源、课程和材料。这些资源帮助开发者更意识到编码实践中的潜在偏见，并提供减轻它们的策略。

2.3 人工智能辅助的管理工具进行软件开发

我们可以使用 JIRA 等 AI 辅助管理工具来规划和跟踪软件开发过程中的各种任务和问题。通过这些工具，团队可以更有效地管理项目进度，分配任务，跟踪问题，并确保项目按计划进行。这些工具不仅提高了团队的协作效率，还帮助团队成员更好地理解和执行项目需求。

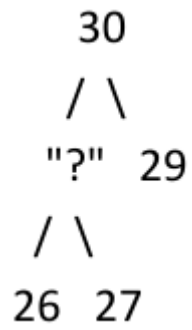
3. 练习

3.1 基础练习

1.给定列表：{2, 9, 5, 4, 8, 1}
冒泡排序算法第一步后，结果是什么?

答案是{2, 5, 4, 8, 1, 9}。因为$2 < 9$，所以不需要交换。

2.对于堆来问号处可以是多少?



答案是28/29。

3.2 进阶练习

3.2.1 编写两个泛型方法来实现归并排序（Merge Sort）

第一个方法使用 Comparable 接口进行排序，第二个方法使用 Comparator 接口进行排序。
示例代码如下。

```
import java.util.Comparator;

public class MergeSortExample {

    // 使用 Comparable 接口进行排序
    public static <E extends Comparable<E>> void mergeSort(E[] list) {
        if (list.length > 1) {
            int middle = list.length / 2;
            @SuppressWarnings("unchecked")
            E[] firstHalf = (E[]) new Comparable[middle];
            @SuppressWarnings("unchecked")
            E[] secondHalf = (E[]) new Comparable[list.length - middle];

            System.arraycopy(list, 0, firstHalf, 0, middle);
            System.arraycopy(list, middle, secondHalf, 0, list.length - middle);

            mergeSort(firstHalf);
            mergeSort(secondHalf);

            merge(list, firstHalf, secondHalf);
        }
    }

    // 使用 Comparator 接口进行排序
    public static <E> void mergeSort(E[] list, Comparator<? super E> comparator)
{
        if (list.length > 1) {
            int middle = list.length / 2;
            @SuppressWarnings("unchecked")
            E[] firstHalf = (E[]) new Comparable[middle];
            @SuppressWarnings("unchecked")
            E[] secondHalf = (E[]) new Comparable[list.length - middle];

            System.arraycopy(list, 0, firstHalf, 0, middle);
            System.arraycopy(list, middle, secondHalf, 0, list.length - middle);
```

```
            mergeSort(firstHalf, comparator);
            mergeSort(secondHalf, comparator);

            merge(list, firstHalf, secondHalf, comparator);
        }
    }

    // 合并两个已排序的子数组
    private static <E extends Comparable<E>> void merge(E[] list, E[] firstHalf,
E[] secondHalf) {
        int i = 0, j = 0, k = 0;
        while (i < firstHalf.length && j < secondHalf.length) {
            if (firstHalf[i].compareTo(secondHalf[j]) <= 0) {
                list[k++] = firstHalf[i++];
            } else {
                list[k++] = secondHalf[j++];
            }
        }
        while (i < firstHalf.length) {
            list[k++] = firstHalf[i++];
        }
        while (j < secondHalf.length) {
            list[k++] = secondHalf[j++];
        }
    }

    // 合并两个已排序的子数组，使用 Comparator
    private static <E> void merge(E[] list, E[] firstHalf, E[] secondHalf,
Comparator<? super E> comparator) {
        int i = 0, j = 0, k = 0;
        while (i < firstHalf.length && j < secondHalf.length) {
            if (comparator.compare(firstHalf[i], secondHalf[j]) <= 0) {
                list[k++] = firstHalf[i++];
            } else {
                list[k++] = secondHalf[j++];
            }
        }
        while (i < firstHalf.length) {
            list[k++] = firstHalf[i++];
        }
        while (j < secondHalf.length) {
            list[k++] = secondHalf[j++];
        }
    }

    public static void main(String[] args) {
        Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
        System.out.println("Original list: " + java.util.Arrays.toString(list));
        mergeSort(list);
        System.out.println("Sorted list using Comparable: " +
java.util.Arrays.toString(list));

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

```
        System.out.println("Original string list: " +
java.util.Arrays.toString(stringList));
        mergeSort(stringList, (a, b) -> b.compareTo(a));
        System.out.println("Sorted string list using Comparator: " +
java.util.Arrays.toString(stringList));
    }
}
```

3.2.2 编写两个泛型方法来实现插入排序（Insertion Sort）

第一个方法使用 Comparable 接口进行排序，第二个方法使用 Comparator 接口进行排序。
示例代码如下。

```
import java.util.Comparator;

public class InsertionSortExample {

    // 使用 Comparable 接口进行排序
    public static <E extends Comparable<E>> void insertionSort(E[] list) {
        for (int i = 1; i < list.length; i++) {
            E current = list[i];
            int j = i - 1;
            while (j >= 0 && list[j].compareTo(current) > 0) {
                list[j + 1] = list[j];
                j--;
            }
            list[j + 1] = current;
        }
    }

    // 使用 Comparator 接口进行排序
    public static <E> void insertionSort(E[] list, Comparator<? super E>
comparator) {
        for (int i = 1; i < list.length; i++) {
            E current = list[i];
            int j = i - 1;
            while (j >= 0 && comparator.compare(list[j], current) > 0) {
                list[j + 1] = list[j];
                j--;
            }
            list[j + 1] = current;
        }
    }

    public static void main(String[] args) {
        Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
        System.out.println("Original list: " + java.util.Arrays.toString(list));
        insertionSort(list);
        System.out.println("Sorted list using Comparable: " +
java.util.Arrays.toString(list));

        String[] stringList = {"banana", "apple", "cherry", "date"};
        System.out.println("Original string list: " +
java.util.Arrays.toString(stringList));
        insertionSort(stringList, (a, b) -> b.compareTo(a));
```

```
        System.out.println("Sorted string list using Comparator: " +
java.util.Arrays.toString(stringList));
    }
}
```