Sorting

Bubble Sort 冒泡排序

Repeatedly steps through the list to be sorted, compares each **pair of adjacent items** and swaps them if they are in the wrong order.

重复遍历要排序的列表，比较**每对相邻项**，如果它们的顺序错误，则交换它们。

但是潜在的情况是如果列表本身就已经是排序好的，但是最初的算法设计中没有检测中断的机制，因此会一直循环下去知道直到到达循环的终止条件，但实际上由于已经排序好了，所以迭代的过程中并没有发生位置转换。因此可以考虑添加一个中断判断。

```
boolean needNextPass = true;
for (int k = 1; k < list.length && needNextPass; k++) {
    // Array may be sorted and next pass not needed
    needNextPass = false;
    for (int i = 0; i < list.length - k; i++){
        if (list[i] > list[i + 1]){
            //swap list[i] with list[i + 1];
            needNextPass = true;
        }
    }
}
```

添加一个参数 **needNextPass**，如果每进入一轮外循环就将值设置为 false，然后如果再当前整个内循环过程中都没有满足过 if (list[i] > list[i + 1])，那么就不会修改 needNextPass，这样就不会开启下一轮的外循环。这样假如列表是是先排序好的，那么实际上只会进行完最开始的一轮遍历之后就结束了

Time complexity of Bubble Sort 冒泡排序的时间复杂度

- **Best case**: Since the number of comparisons is **n - 1** in the first pass, the best- case time for a bubble sort is **O(n)**.

  最佳情况：由于第一次比较的次数为n - 1，因此冒泡排序的最佳时间为**O(n)**。

- **Worst case**: example: Initial list: [5,4,3,2,1]

  最差情况

  - [5,4,3,2,1] -> [4,3,2,1,5] - 1st pass, 4 comparsions (5 v.s. 4,3,2,1, 5 is sorted)
  - [4,3,2,1,5] -> [3,2,1,4,5] - 2nd pass, 3 comparsions (4 v.s. 3,2,1, 4 is sorted)
  - [3,2,1,4,5] -> [2,1,3,4,5] - 3rd pass, 2 comparsions (3 v.s. 1,2, 3 is sorted)
  - [2,1,3,4,5] -> [1,2,3,4,5] - 4th pass, 1 comparisons (2 v.s. 1, 2 is sorted)
  - So, 5 elements, we do 4+3+2+1 comparisons
    - If **n** elements, in the worst case, we compare **(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2** times. The Big(O) is **O(n^2)**

```
public class BubbleSort {
    public static void bubbleSort(int[] list) {
```

```java
        boolean needNextPass = true;
        for (int k = 1; k < list.length && needNextPass; k++) {
            // Array may be sorted and next pass not needed
            needNextPass = false;
            // Perform the kth pass
            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; // Next pass still needed
                }
            }
        }
    }

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

    // 随机生成一个数组
    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
}
```

Merge Sort 归并排序

The merge-sort algorithm can be described recursively as follows:

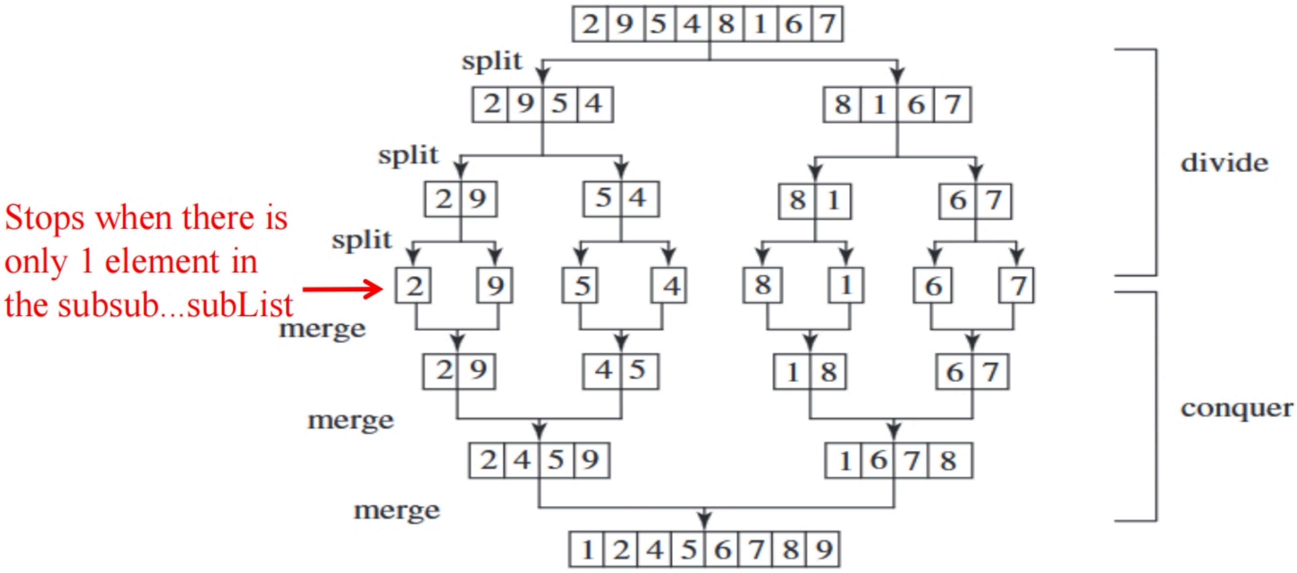
合并排序算法可以递归地描述如下：

- The algorithm divides the array into two halves and applies a merge sort on each half recursively.

  该算法将数组分为两半，并递归地对每一半应用合并排序。

- After the two halves are sorted, the algorithm then merges them.

  在对两半部分进行排序后，算法会将它们合并。




```java
public class MergeSortTest {
    // 拆分排序
    public static void mergeSort(int[ ] list) {
        if (list.length > 1) { // Recursive base case: stop when condition unsatisfied
            // Split the 1st half (recursive step)
            int[] firstHalf = new int[list.length / 2];
            System.arraycopy(list,0,firstHalf,0,list.length / 2);
            mergeSort(firstHalf);
            // Split the 2nd half (recursive step)
            int secondHalfLength = list.length - list.length / 2;
            int[ ] secondHalf = new int[secondHalfLength] ;
            System.arraycopy(list,list.length / 2 ,secondHalf, 0 ,
                             secondHalfLength);
            mergeSort(secondHalf);
            // SortMerge (only happens AFTER both recursive calls
            finish)
                merge( firstHalf, secondHalf, list) ;
        }
    }

    // 合并
    public static void merge(int[] list1, int[] list2, int[] temp){
        int current1 = 0; // Current index in list1, the first half
        int current2 = 0; // Current index in list2, the 2nd half
        int current3 = 0; // Current index in temp, storing data temporarily
        // While the indices are in the list
        while (current1 < list1.length && current2 < list2.length) {
            if (list1[current1] < list2[current2])
                // If current element in list1 is smaller, add it to temp
                temp[current3++] = list1[current1++];
            else
                // Otherwise, add the current element in list2 to temp
                temp[current3++] = list2[current2++];
        }
        // list2 finished, but there are remaining elements in list1, add
```

```
        them to temp
            while (current1 < list1.length)
                temp[current3++] = list1[current1++];
        // list1 finished, but there are remaining elements in list2, add
        them to temp
            while (current2 < list2.length)
                temp[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);
    }
}
```

Time complexity of merge sort 归并排序的时间复杂度

- Let **T(n)** denote the time required for sorting an array of **n** elements using merge sort.

  设T（n）表示使用合并排序对n个元素数组进行排序所需的时间。

- $$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 2n - 1$$

- The first **T(n/2)** is the time for sorting the first half of the array and the second **T(n/2)** is the time for sorting the second half

  第一个 T（n/2） 是对数组的前半部分进行排序的时间，第二个 T（n/2） 是对后半部分进行排序的时间

- Merging cost **2n-1** because **n - 1** comparisons (for comparing the elements of the two subarrays) and **n** moves (to place each element into the temporary array)

  合并成本为 2n-1，因为 n 1 比较（用于比较两个子数组的元素）和 n 移动（将每个元素放入临时数组中）

- Repeatedly substitute T(n/2) into the formula we will find the time complexity of merge sort is **O(nlogn)**.

  在公式中反复代入 T（n/2），我们会发现归并排序的时间复杂度为 **O（nlogn）**

Quick Sort 快速排序

- A quick sort works as follows:

  快速排序的工作原理如下

  - The algorithm selects an element, called the **pivot**, in the array.

    该算法在数组中选择一个名为 **枢轴** 的元素。

- It partitions (divides) the array into two parts so all the elements in the **first part** are less than or equal to the pivot, and all the elements in the **second part** are greater than the pivot.

  它将数组划分为两部分，因此 **第一部分** 中的所有元素都小于或等于枢轴，**第二部分** 中所有元素都大于枢轴。

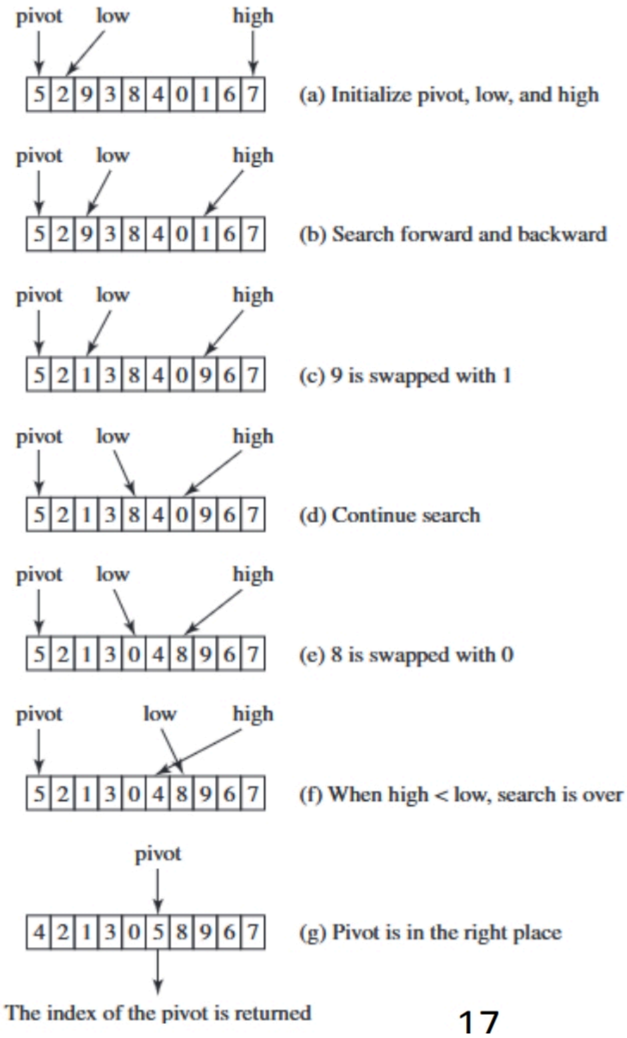
- The quick-sort algorithm is then **recursively** applied to the first part and then the second part to sort them out.

  然后 **递归地** 将快速排序算法应用于第一部分，然后再应用于第二部分以对其进行排序。


```
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];
        int low = first + 1; // position, not value
        int high = last; // position, not value
        while (high > low) {
            //low pointer moves right when
            //1. low <=high and 2. list[low] <= pivot
            while (low <= high && list[low] <= pivot)
                low++;
            //high pointer moves left when
            //1. low <=high and 2. list[high] > pivot
            while (low <= high && list[high] > pivot)
                high--;
            // If low < high, swap the two elements.
            if (high > low) {
                int temp = list[high];
                list[high] = list[low];
                list[low] = temp;
            }
        }

        // Ensure high pointer point at an element
        // less than or equal to pivot
        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 // e.g., a sotred list
            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);
    }
}
```

这个 quickSort() 定义了 Quick Sort 的递归结构，因为它不断调用自身。partition() 也会在进程中递归调用，直到基本情况 （last>first)。这确保了列表可以递归地分为左和右子列表、子子列表、子子子列表……



Time Complexity of Quick Sort

- To partition an array of **n** elements, it takes **n** comparisons and **n** moves. Thus, the time required for partition is **O(n)**.

  要对一个包含 **n** 个元素的数组进行分区，需要进行 **n** 次比较和 **n** 次移动。因此，分区所需的时间是 **O（n）**。

- In the **best case,** each time the pivot divides the array into two parts of the same size. **O(nlogn)**

  在 **最好的情况**下， 每次pivot将数组分成大小相同的两部分。**O（nlogn）**

- In the **average case**, maybe not exactly the same, but the size of the two sub arrays are very close. **O(nlogn)**

  在 **平均情况** 下，可能不完全相同，但两个子数组的大小非常接近 **O（nlogn）**

- In the **worst case**, the pivot divides the array each time into one big subarray **with the other array empty**. The size of the big subarray is one less than the one before divided. **O(n^2)**

  在 **最坏的情况** 下，枢轴每次将数组划分为一个大子数组，另一个数组为空。大子阵列的大小比分割前的小一个 **O(n^2)**

  - For example (assuming the 1st element is the pivot for partition):

    例如（假设第一个元素是分区的枢轴）：

    - [1,2,3,4,5...n], size=n
    - In the first partition, pivot=1, we have ->

        - left: [empty], right [2,3,4,5...n], right sub array size = n-1
    - In the next partition, pivot=2, we have ->
        - left: [empty], right [3,4,5,6...n], right sub array size = n-2
    - ... continues this way, we will have to recursively divide the arrary **n-1** times till the sub array size =1
    - Recall that the time complexity of each partition is O(n), as it compares all elements during each division.
    - So we did **n-1** times **O(n)**, so, we get the **worst time complexity** to be **O(n^2)**

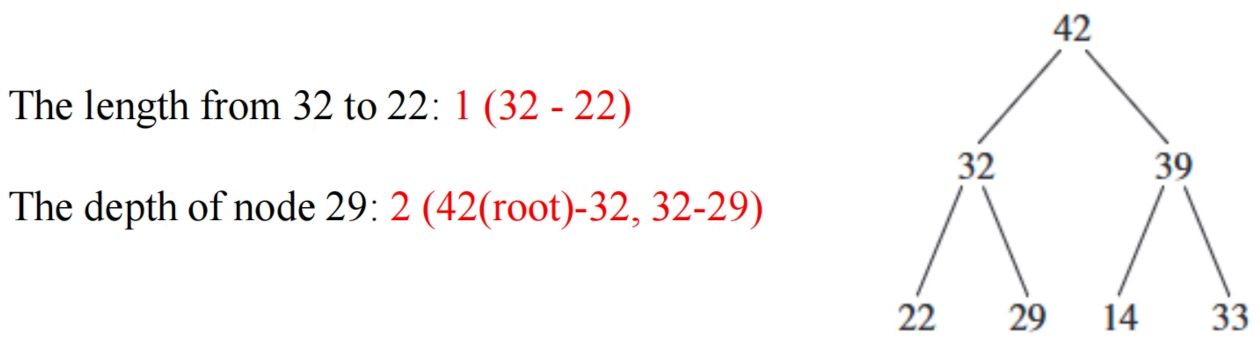
Heap Sort: Binary Tree 堆排序

- A **binary tree** is a hierarchical structure: it either is empty or it consists of an element, called the **root**, and two distinct binary trees, called the **left subtree** and **right subtree**

  二叉树 是一种层次结构：它要么是空的，要么由一个元素（称为 **根** ）和两个不同的二叉树（称为 **左子树** 和 **右子树**）组成。
    - The **length** of a path is the number of the edges in the path

      路径的 **长度** 是路径中的边数
    - The **depth** of a node is the length of the path from the root to that node

      节点的 **深度** 是从根节点到该节点的路径长度



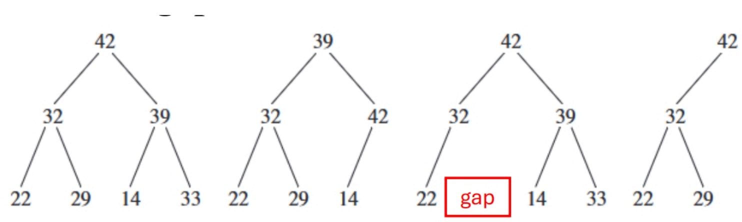
Complete Binary Tree 完全二叉树

A binary tree is **complete** if:

- All levels are completely filled except **possibly** the last level

  除可能最后一级外，所有上层级别都已完全填满
- Even though the last level may not be full, it must be filled from left to right, **without gaps**

  即使最后一关可能没有填满，也必须从左到右填满，**没有空隙**



- 1st - complete, as all levels are completely filled

因为所有级别都已完全填满

- 2nd - complete, although the last level is not full (missing one node next to 14), it is filled from left to right, without gaps

  尽管最后一级没有填满（缺少14旁边的一个节点），但它是从左到右填充的，没有间隙

- 3rd, incomplete, a gap from left to right (next to node 22)

  从左到右存在间隙（在节点 22 旁边）

- 4th, incomplete, the 2nd level is not full, while we only allow the last level (level 3 in this case) not to be full

  第二级未满，而我们只允许最后一级（本例中为第3级）未满

Binary Heap 二叉堆

- A **binary heap** is a binary tree with the following properties:

  二进制堆是具有以下属性的二叉树

  - It is a **complete binary tree**, and

    它是一个 完全二叉树

  - **Each node** is greater than or equal to any of its children

    每个节点都**大于或等于**其任何**子节点**

Heap Sort 堆排序

- **Heap sort** uses a binary heap and the process consists of **two** main phases:

  堆排序使用二叉堆，该过程由**两个**主要阶段组成：

  - **Heap construction:**

    堆构造：

    - All the elements are first inserted into a max heap

      首先将所有元素插入到最大堆中

  - **Repeated removal:**

    重复移除：

    - Repeatedly remove the **root node**, which is the current largest element in the heap. The removed element is actually moved to the end of the array, forming a sorted array that grows from the back.

      重复删除 **根**节点，这是堆中当前最大的元素。删除的元素实际上被移动到数组的末尾，形成一个从后面增长的排序数组。

Example:

- For example: [10,5,3,4,1]
  - 1st removal: [..., 10]
  - 2nd removal: [..., 5, 10]
  - 3rd removal: [..., 4, 5, 10]
  - (do removal repeatedly)
  - Final: [1,3,4,5,10]

Sorting a Heap 对一个堆进行排序

- A heap can be stored in an **ArrayList** or an **array** if the heap size is known in advance

  如果堆大小预先已知，则可以将堆存储在 **ArrayList** 或 **数组** 中

  - For a node at position **i**, its left child is at position **2i+1** and its right child is at position **2i+2**, and its parent is at index **(i-1)/2**

    对于位于位置 **i** 的节点，其左子节点位于位置 **2i+1**，右子节点位于地址 **2i+2**，其父节点位于索引 **（i-1）/2**

  - For example: the for a nood at position **4**, and its two children are at positions 2 * 4+1=**9** and 2 * 4+2 =**10**, and its parent is at index (4-1)/2=**1** **(not 1.5, integer division)**

    举例来说：对于位置为 4 节点，其两个子节点位于位置2 * 4+1=9 和2 * 4+2 =10，其父节点位于索引（4-1）/2=1 （不是1.5，整数除法，向下取整）

Adding elements to a Heap 向堆中添加一个元素

- To add a new node to a heap, first add it to the end of the heap and then rebuild the tree with this algorithm:

  要将新节点添加到堆中，请先将其添加到堆的末尾，然后使用以下算法重建树：

```
// Let the last node be the current node;
while (the current node is greater than its parent) {
    Swap the current node with its parent;
    Now the current node is one level up;
}
```

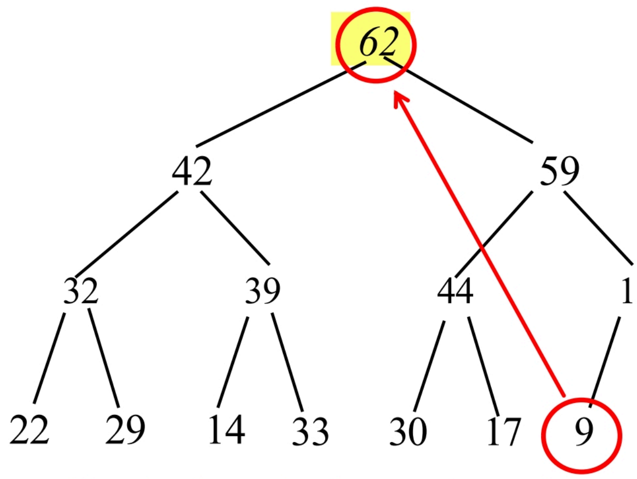
Removing the Root and Rebuild the Heap 从堆中移除根并重构堆

- Often we need to remove the maximum element, which is the **root in a heap**
- After the root is removed, the tree must be rebuilt to **maintain the heap property (e.g., parent>=child)** using this algorithm:

```
Move the last node to replace the root;
Let the root be the current node;
while (the current node has children and the
       current node is smaller than one of its children)
{ Swap the current node with the larger of its
    children;
 Now the current node is one level down;
}
```

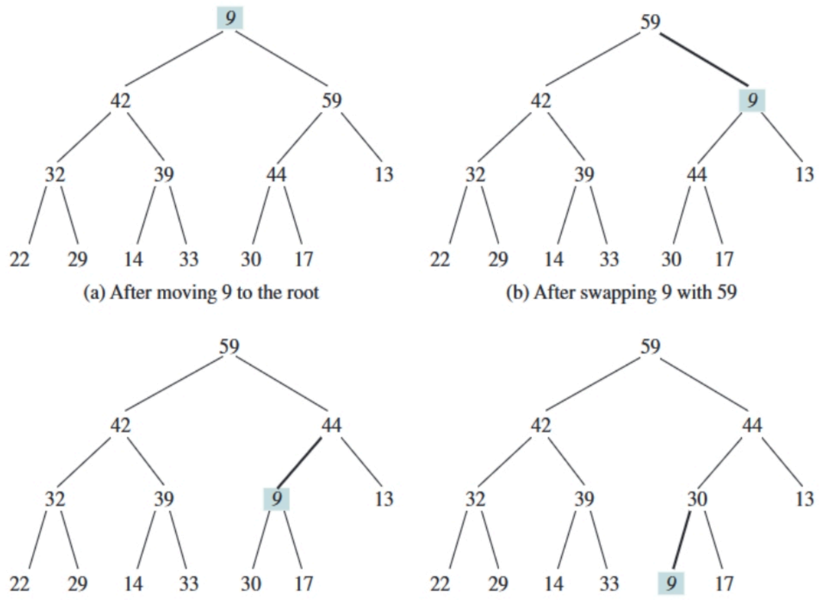
Example:

- Removing root 62 from the heap (replaces it with the last node in the heap: 9)



Move the last node to replace the root;
Let the root be the current node;

```
while (the current node has children and the
       current node is smaller than one of its children)
{ Swap the current node with the larger of its
    children;
}
```



(a) After moving 9 to the root

(b) After swapping 9 with 59

(c) After swapping 9 with 44

(d) After swapping 9 with 30

(a-b) 9<42; 9<59; 59>42;
Swap 9 and 59

(b-c) 9<44; 9<13; 44>13;
Swap 9 and 44

(c-d) 9<30; 9<17; 30>17;
Swap 9 and 30

The heap is rebuilt now, and ready for the next removal

```
public class Heap<E extends Comparable> {
    private java.util.ArrayList<E> list = new java.util.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) {
```

```java
        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();
    }
}

public class HeapSort {
```

```java
    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] + " ");
    }
}
```

Time complexity of Heap Sort 堆排序的时间复杂度

Heap SortTime: **O(nlog n)**

- **Space Complexity**
    - Both merge and heap sorts require O(n logn) time.

      合并和堆排序都需要O（n logn）时间。
    - A merge sort requires a temporary array for merging two subarrays; a heap sort does not need additional array space.

      合并排序需要一个临时数组来合并两个子数组；堆排序不需要额外的数组空间。
    - Therefore, a heap sort is more **space efficient** than a merge sort.

      因此，**堆排序比合并排序更节省空间**。