

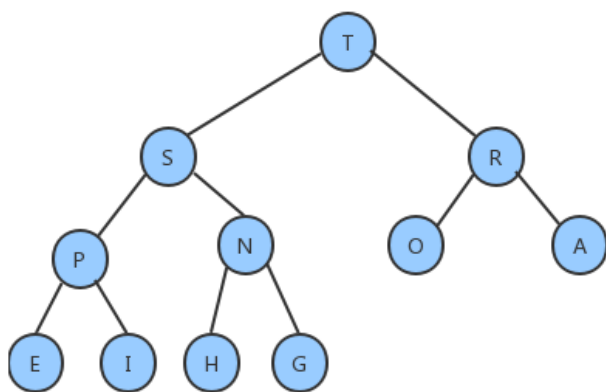
一、堆

1.1 堆的定义

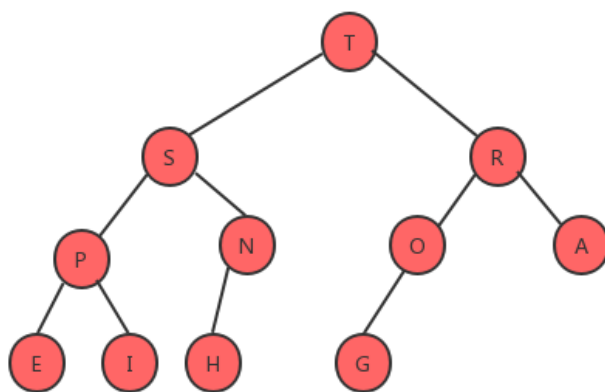
堆是计算机科学中一类特殊的数据结构的统称，堆通常可以被看做是一棵完全二叉树的数组对象。

堆的特性：

1.它是完全二叉树，除了树的最后一层结点不需要是满的，其它的每一层从左到右都是满的，如果最后一层结点不是满的，那么要求左满右不满。



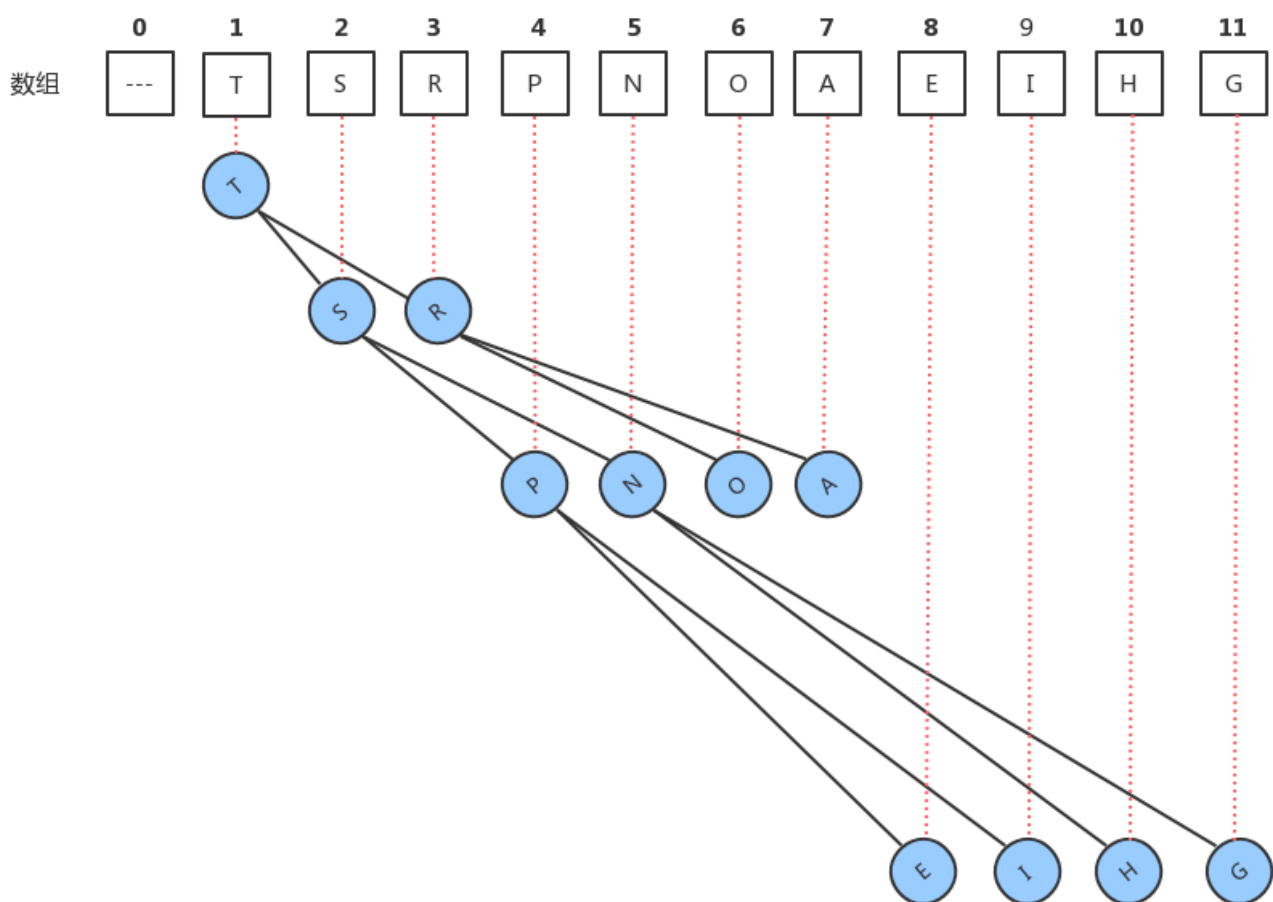
完全二叉树



不完全二叉树

2.它通常用数组来实现。

具体方法就是将二叉树的结点按照层级顺序放入数组中，根结点在位置1，它的子结点在位置2和3，而子结点的子结点则分别在位置4,5,6和7，以此类推。



如果一个结点的位置为 k ，则它的父结点的位置为 $\lfloor k/2 \rfloor$ ，而它的两个子结点的位置则分别为 $2k$ 和 $2k+1$ 。这样，在不使用指针的情况下，我们也可以通过计算数组的索引在树中上下移动：从 $a[k]$ 向上一层，就令 k 等于 $k/2$ ，向下一层就令 k 等于 $2k$ 或 $2k+1$ 。

3. 每个结点都大于等于它的两个子结点。这里要注意堆中仅仅规定了每个结点大于等于它的两个子结点，但这两个子结点的顺序并没有做规定，跟我们之前学习的二叉查找树是有区别的。

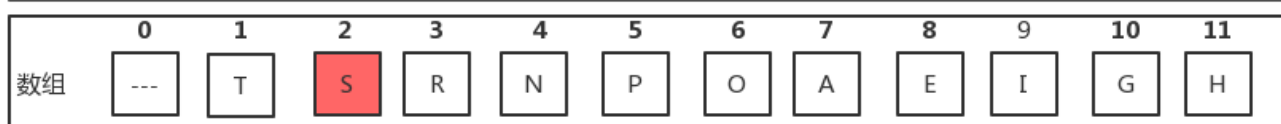
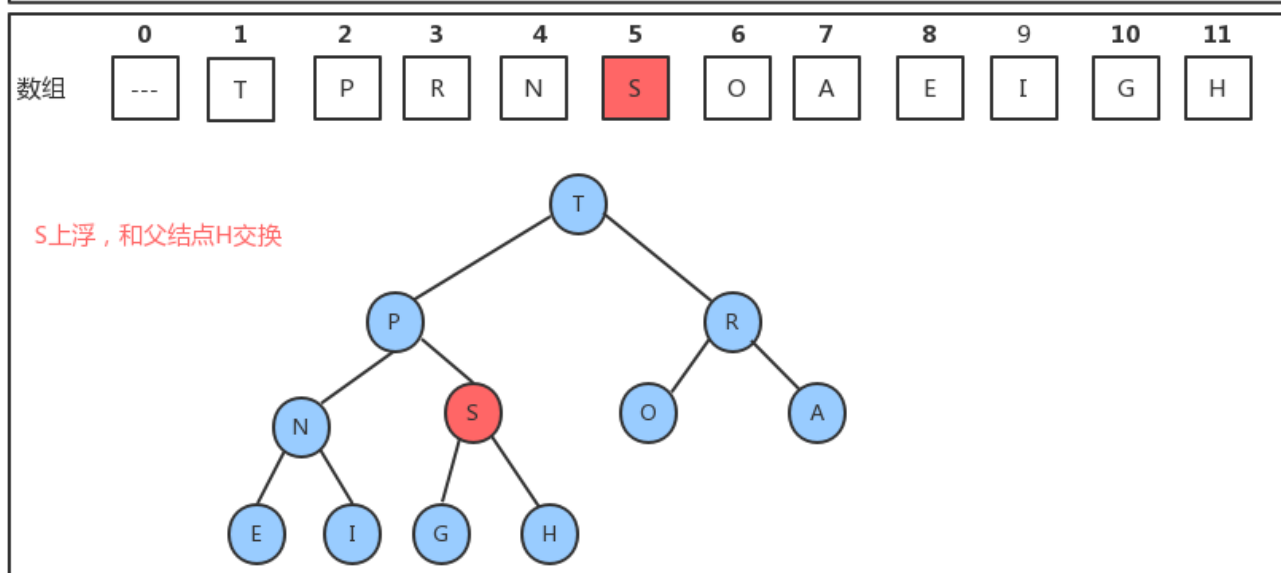
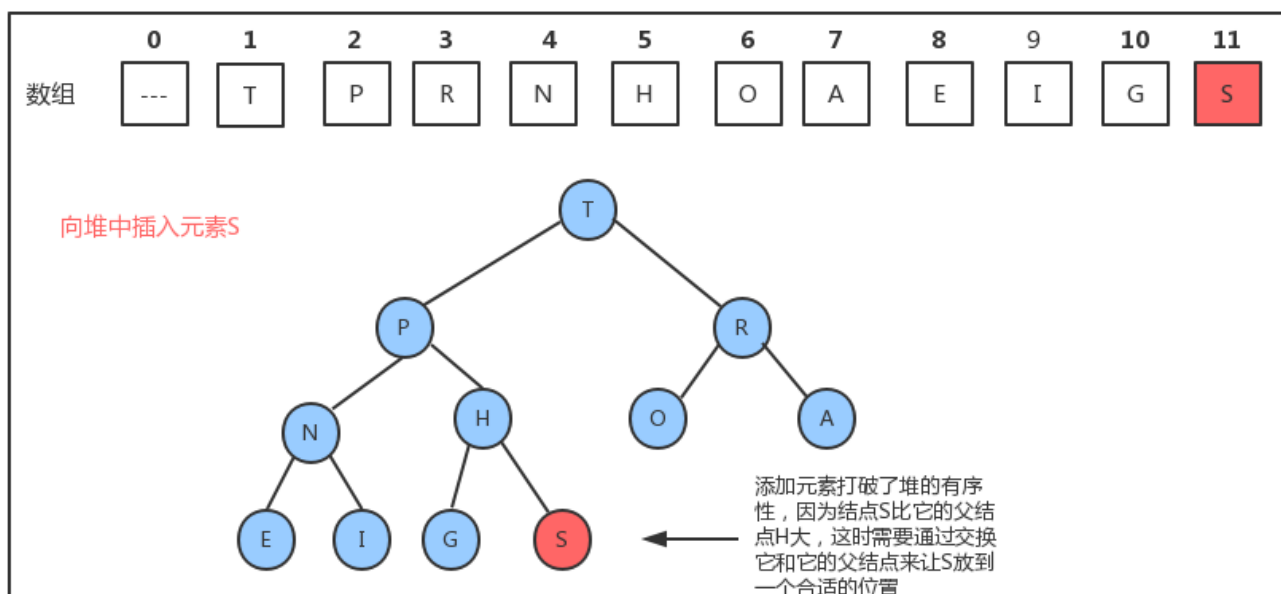
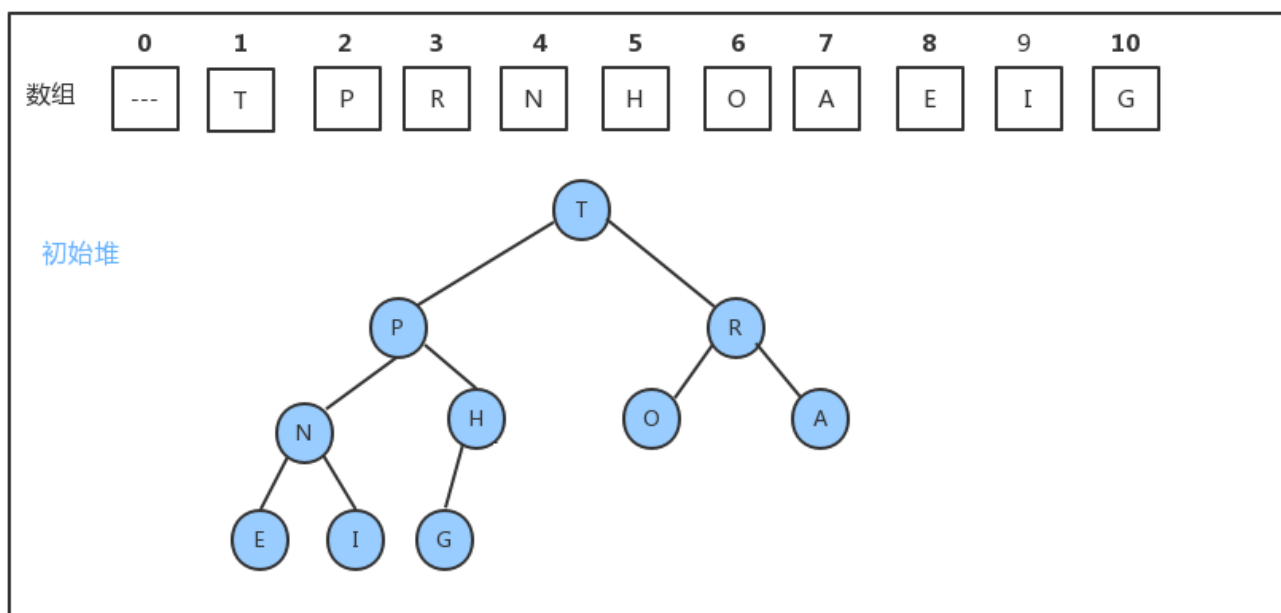
1.2 堆的API设计

类名	Heap>
构造方法	Heap(int capacity)：创建容量为capacity的Heap对象
成员方法	1.private boolean less(int i,int j)：判断堆中索引i处的元素是否小于索引j处的元素 2.private void exch(int i,int j):交换堆中i索引和j索引处的值 3.public T delMax():删除堆中最大的元素,并返回这个最大元素 4.public void insert(T t)：往堆中插入一个元素 5.private void swim(int k):使用上浮算法，使索引k处的元素能在堆中处于一个正确的位置 6.private void sink(int k):使用下沉算法，使索引k处的元素能在堆中处于一个正确的位置
成员变量	1.private T[] imtes：用来存储元素的数组 2.private int N：记录堆中元素的个数

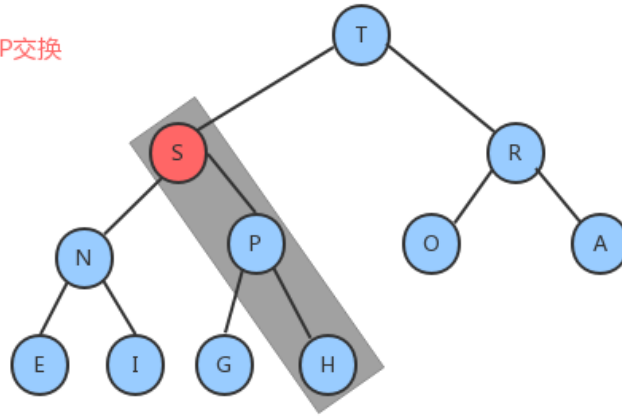
1.3 堆的实现

1.4.1 insert插入方法的实现

堆是用数组完成数据元素的存储的，由于数组的底层是一串连续的内存地址，所以我们要往堆中插入数据，我们只能往数组中从索引0处开始，依次往后存放数据，但是堆中对元素的顺序是有要求的，每一个结点的数据要大于等于它的两个子结点的数据，所以每次插入一个元素，都会使得堆中的数据顺序变乱，这个时候我们就需要通过一些方法让刚才插入的这个数据放入到合适的位置。



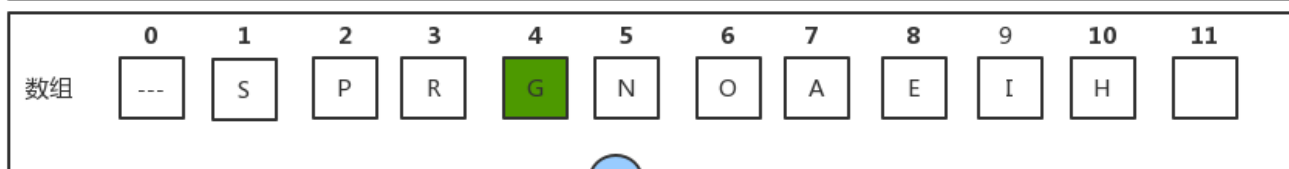
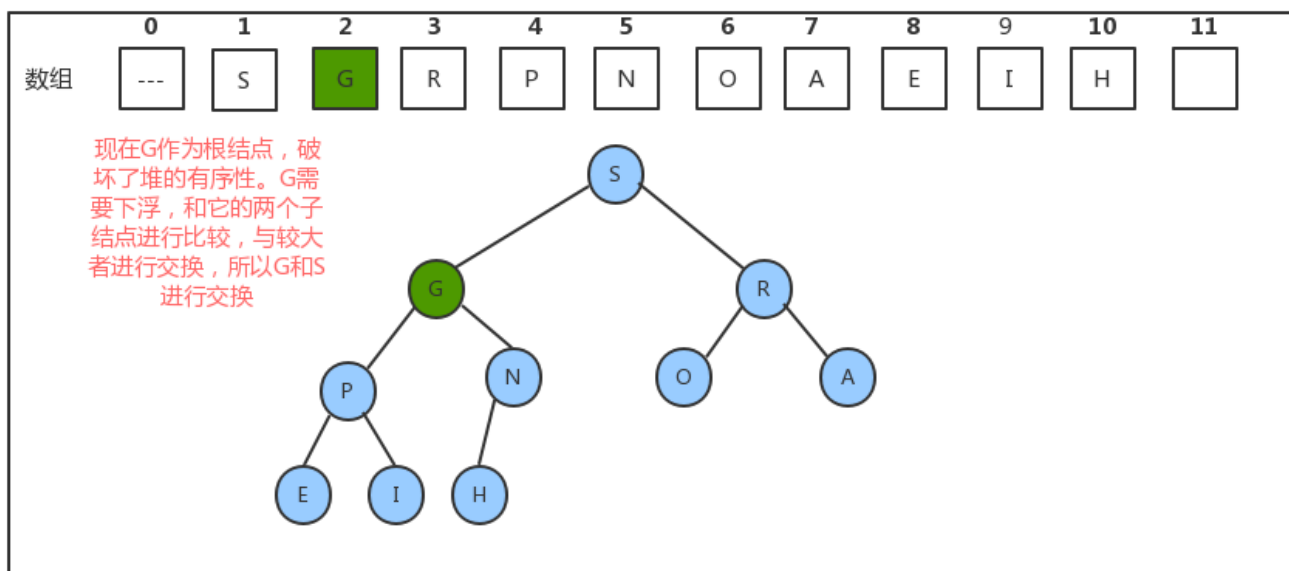
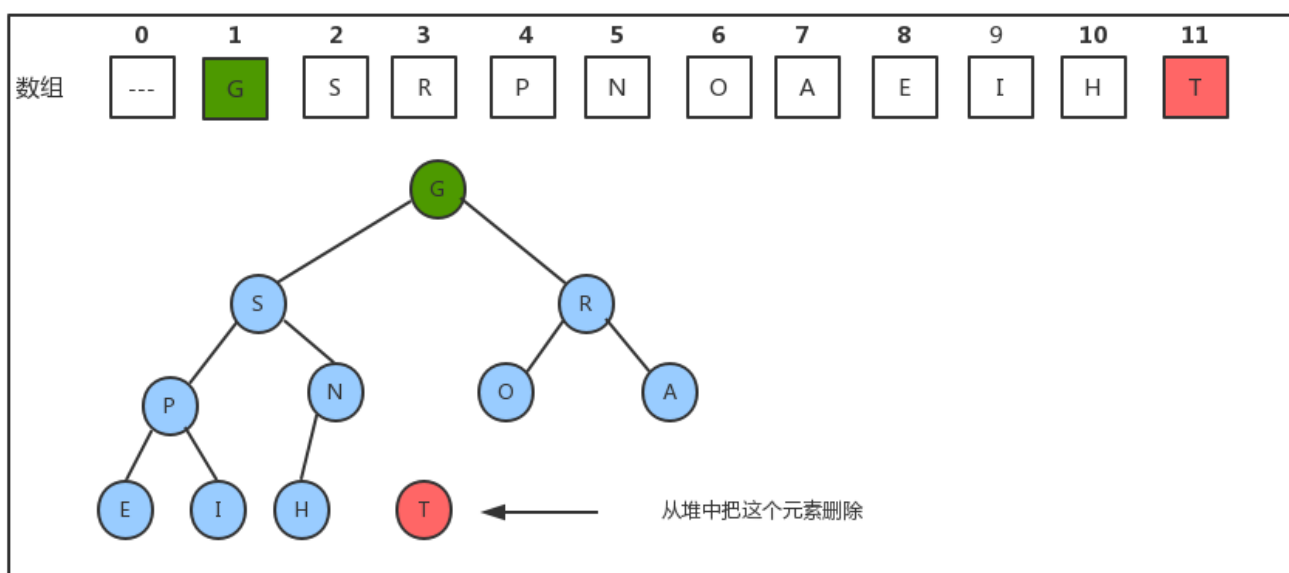
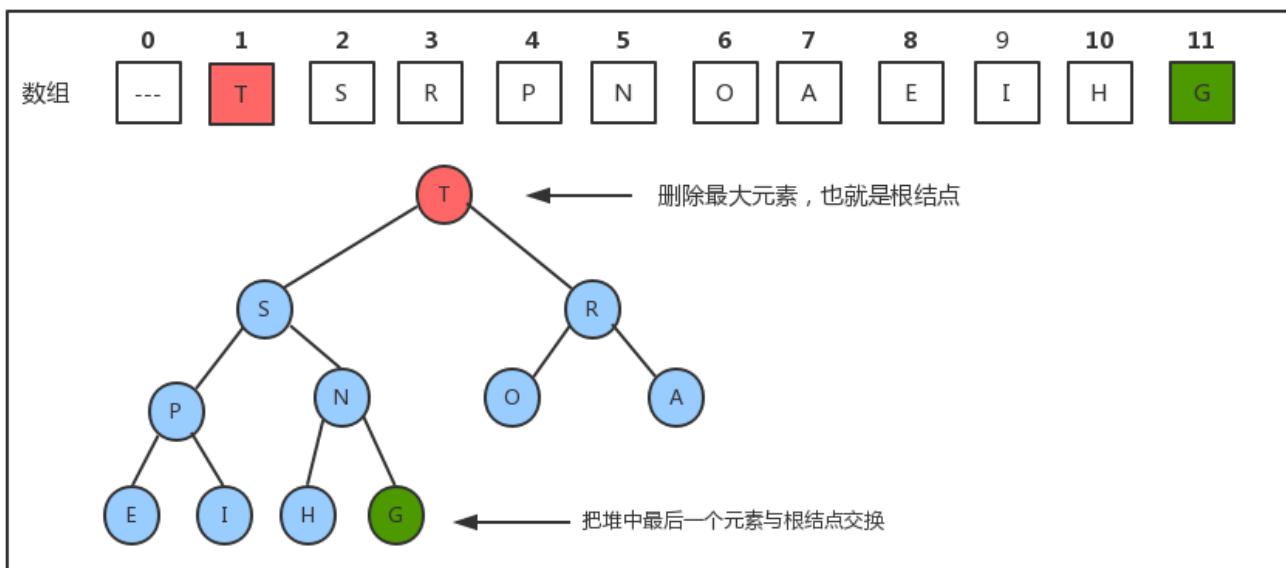
S上浮，和父结点P交换

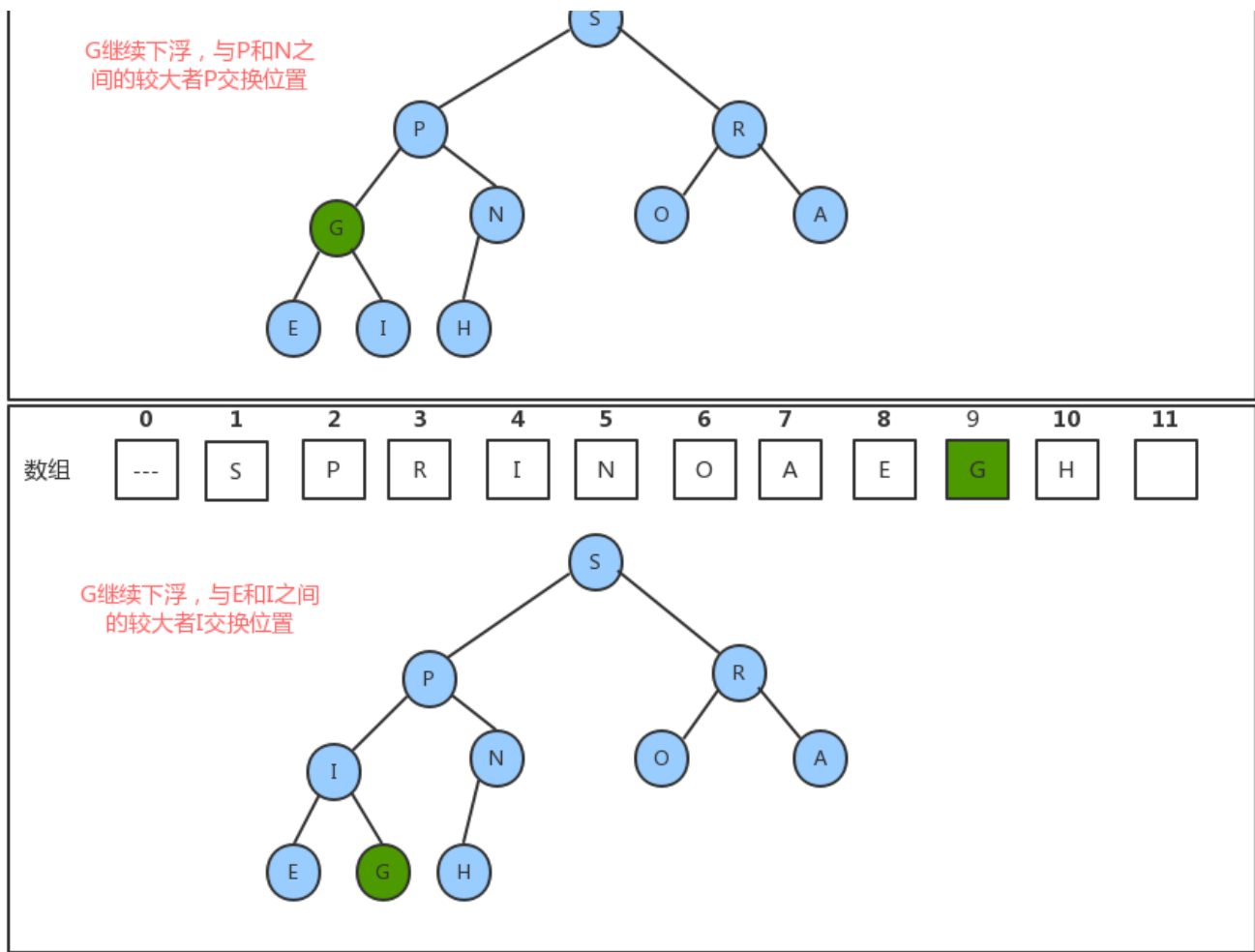


所以，如果往堆中新插入元素，我们只需要不断的比较新结点 $a[k]$ 和它的父结点 $a[k/2]$ 的大小，然后根据结果完成数据元素的交换，就可以完成堆的有序调整。

1.4.2 delMax删除最大元素方法的实现

由堆的特性我们可以知道，索引1处的元素，也就是根结点就是最大的元素，当我们把根结点的元素删除后，需要有一个新的根结点出现，这时我们可以暂时把堆中最后一个元素放到索引1处，充当根结点，但是它有可能不满足堆的有序性需求，这个时候我们就需要通过一些方法，让这个新的根结点放入到合适的位置。





所以，当删除掉最大元素后，只需要将最后一个元素放到索引1处，并不断的拿着当前结点 $a[k]$ 与它的子结点 $a[2k]$ 和 $a[2k+1]$ 中的较大者交换位置，即可完成堆的有序调整。

1.4.3 堆的实现代码

```

1  //堆代码
2  public class Heap<T extends Comparable<T>> {
3      //存储堆中的元素
4      private T[] items;
5      //记录堆中元素的个数
6      private int N;
7
8      public Heap(int capacity) {
9          items = (T[]) new Comparable[capacity+1];
10         N=0;
11     }
12
13     //判断堆中索引i处的元素是否小于索引j处的元素
14     private boolean less(int i,int j){
15         return items[i].compareTo(items[j])<0;
16     }
17
18     //交换堆中i索引和j索引处的值
19     private void exch(int i,int j){

```

```

20     T tmp = items[i];
21     items[i] = items[j];
22     items[j] = tmp;
23 }
24
25
26
27 //往堆中插入一个元素
28 public void insert(T t){
29     items[++N] = t;
30     swim(N);
31 }
32
33 //删除堆中最大的元素,并返回这个最大元素
34 public T delMax(){
35     T max = items[1];
36     //交换索引1处和索引N处的值
37     exch(1,N);
38     //删除最后位置上的元素
39     items[N]=null;
40     N--;//个数-1
41     sink(1);
42     return max;
43 }
44
45 //使用上浮算法,使索引k处的元素能在堆中处于一个正确的位置
46 private void swim(int k){
47     //如果已经到了根结点,就不需要循环了
48     while(k>1){
49         //比较当前结点和其父结点
50         if(less(k/2,k)){
51             //父结点小于当前结点,需要交换
52             exch(k/2,k);
53         }
54         k = k/2;
55     }
56 }
57
58 //使用下沉算法,使索引k处的元素能在堆中处于一个正确的位置
59 private void sink(int k){
60     //如果当前已经是最底层,就不需要循环了
61     while(2*k<=N){
62         //找到子结点中的较大者
63         int max;
64         if (2*k+1<=N){//存在右子结点
65             if (less(2*k,2*k+1)){
66                 max = 2*k+1;
67             }else{
68                 max = 2*k;
69             }
70         }else{//不存在右子结点
71             max = 2*k;
72         }

```



```

73
74         //比较当前结点和子结点中的较大者，如果当前结点不小，则结束循环
75         if (!less(k,max)){
76             break;
77         }
78         //当前结点小，则交换，
79         exch(k,max);
80         k = max;
81     }
82 }
83 }
84
85 //测试代码
86 public class Test {
87     public static void main(String[] args) throws Exception {
88         Heap<String> heap = new Heap<String>(20);
89         heap.insert("S");
90         heap.insert("G");
91         heap.insert("I");
92         heap.insert("E");
93         heap.insert("N");
94         heap.insert("H");
95         heap.insert("O");
96         heap.insert("A");
97         heap.insert("T");
98         heap.insert("P");
99         heap.insert("R");
100
101         String del;
102         while((del=heap.delMax())!=null){
103             System.out.print(del+",");
104         }
105
106     }
107 }

```

1.4 堆排序

给定一个数组：

```
String[] arr = {"S","O","R","T","E","X","A","M","P","L","E"}
```

请对数组中的字符按从小到大排序。

实现步骤：

- 1.构造堆；
- 2.得到堆顶元素，这个值就是最大值；
- 3.交换堆顶元素和数组中的最后一个元素，此时所有元素中的最大元素已经放到合适的位置；
- 4.对堆进行调整，重新让除了最后一个元素的剩余元素中的最大值放到堆顶；
- 5.重复2~4这个步骤，直到堆中剩一个元素为止。

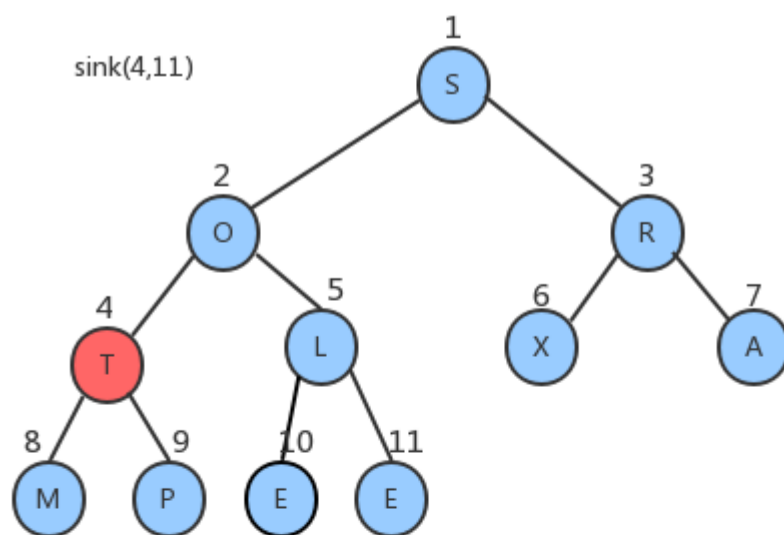
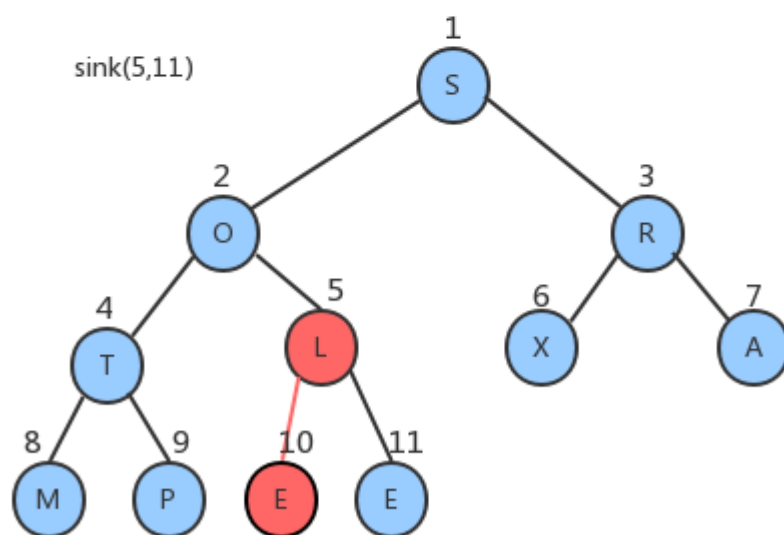
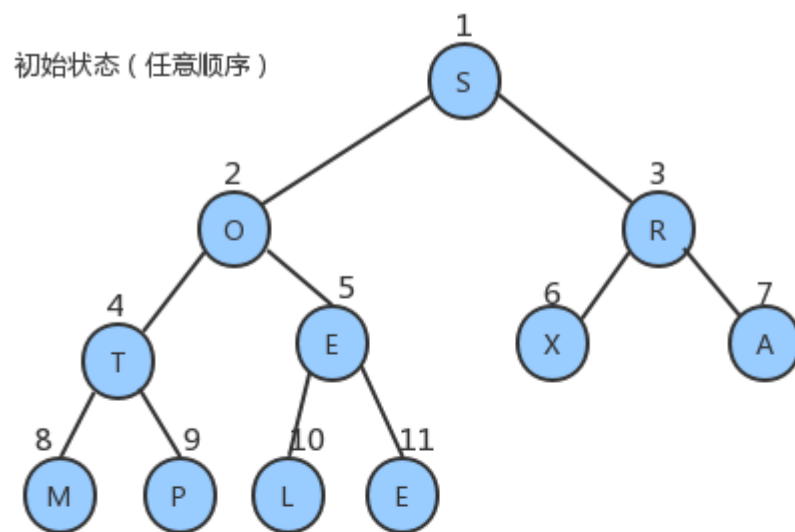
API设计：

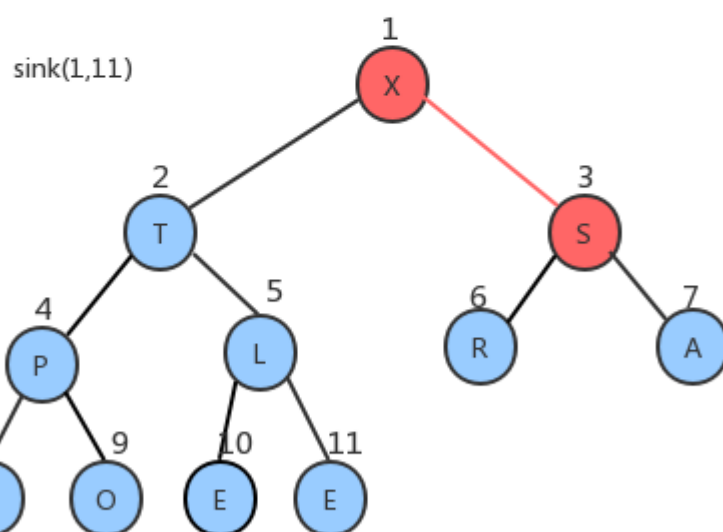
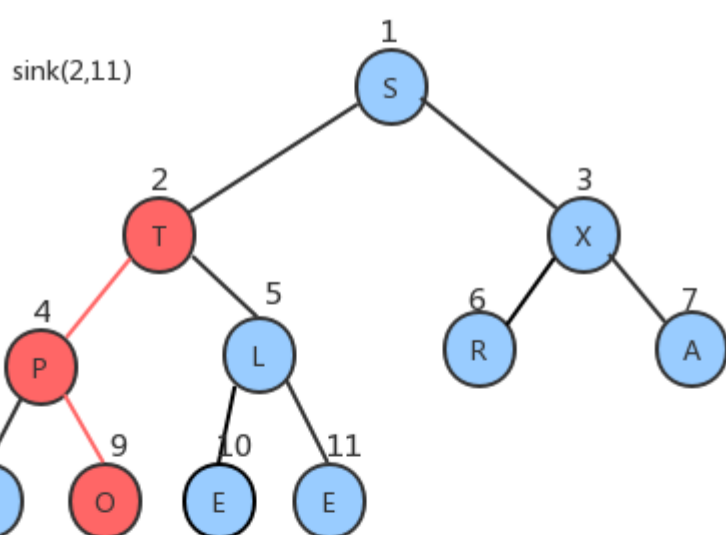
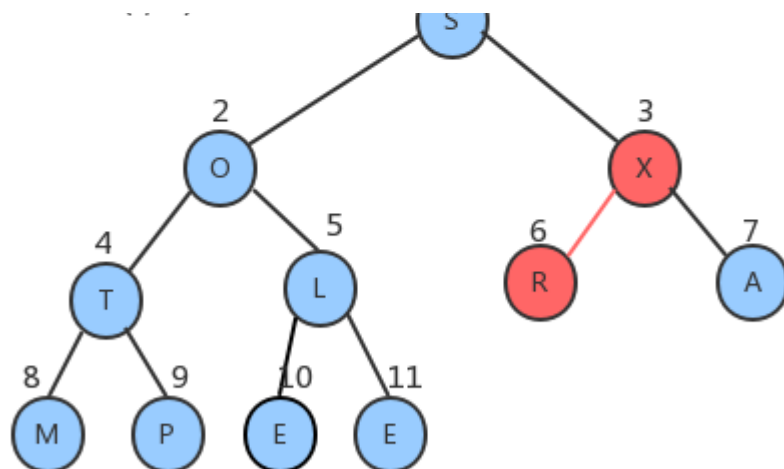
类名	HeapSort>
成员方法	1.public static void sort(Comparable[] source)：对source数组中的数据从小到大排序 2.private static void createHeap(Comparable[] source, Comparable[] heap):根据原数组source，构造出堆heap 3.private static boolean less(Comparable[] heap, int i, int j)：判断heap堆中索引i处的元素是否小于索引j处的元素 4.private static void exch(Comparable[] heap, int i, int j):交换heap堆中i索引和j索引处的值 5.private static void sink(Comparable[] heap, int target, int range):在heap堆中，对target处的元素做下沉，范围是0~range。

1.4.1 堆构造过程

堆的构造，最直观的想法就是另外再创建一个和新数组数组，然后从左往右遍历原数组，每得到一个元素后，添加到新数组中，并通过上浮，对堆进行调整，最后新的数组就是一个堆。

上述的方式虽然很直观，也很简单，但是我们可以用更聪明一点的办法完成它。创建一个新数组，把原数组0~length-1的数据拷贝到新数组的1~length处，再从新数组长度的一半处开始往1索引处扫描（从右往左），然后对扫描到的每一个元素做下沉调整即可。



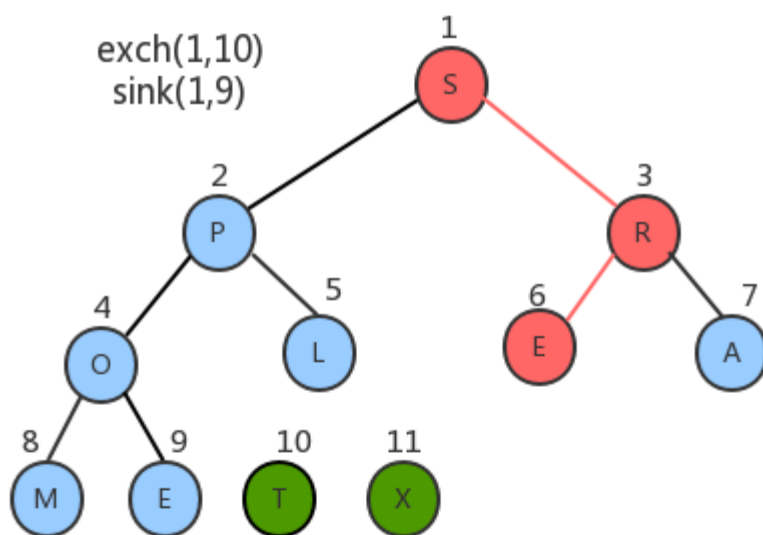
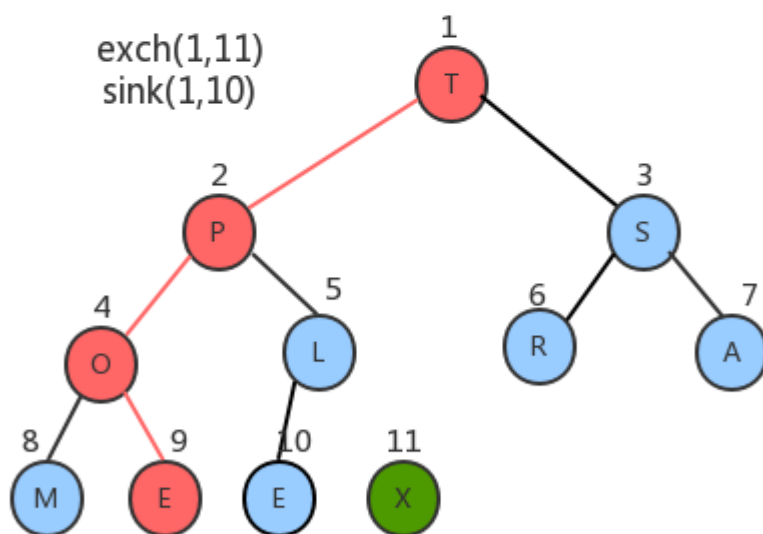
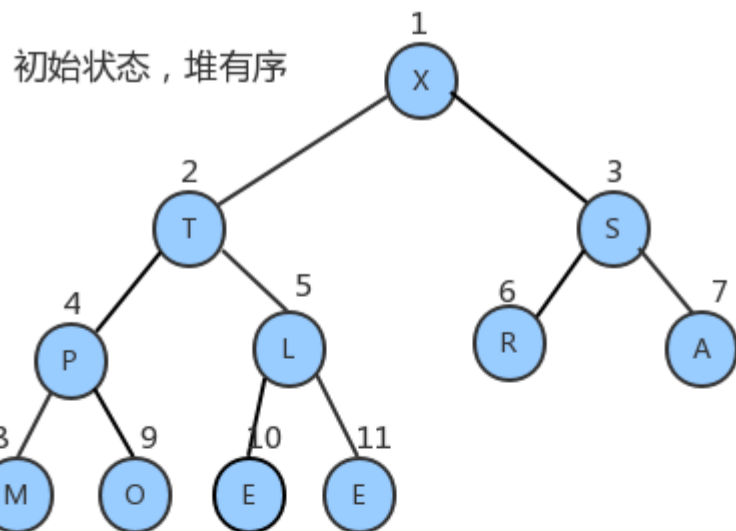


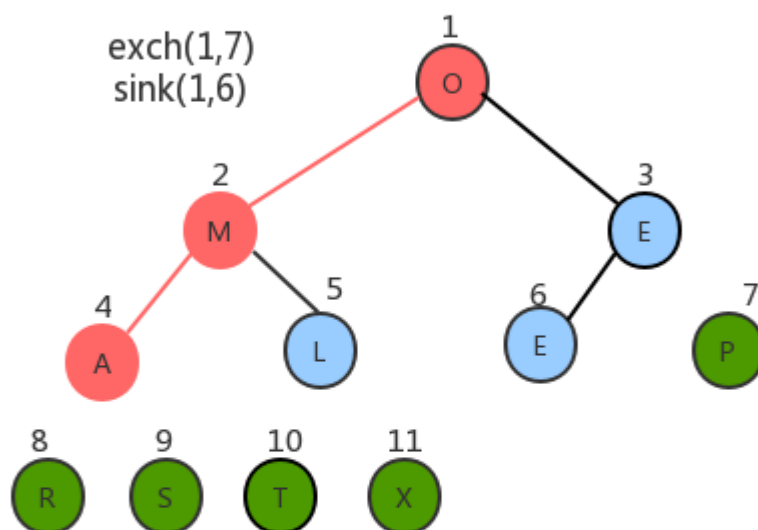
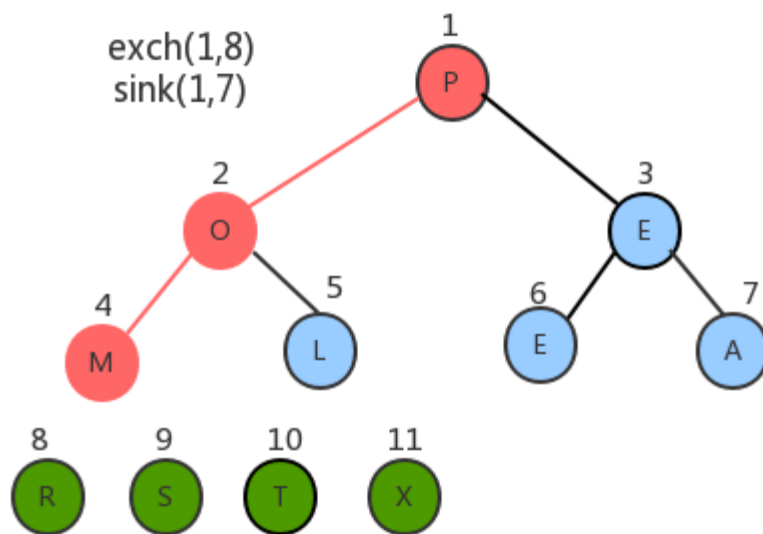
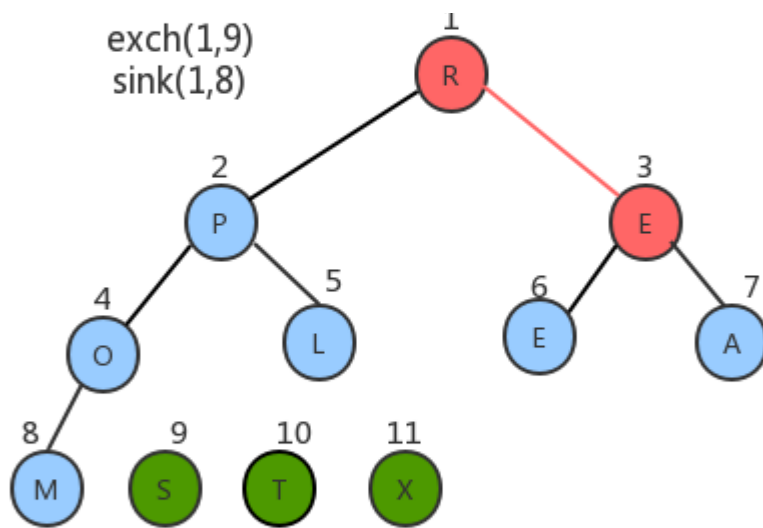
堆构造完毕，堆有序

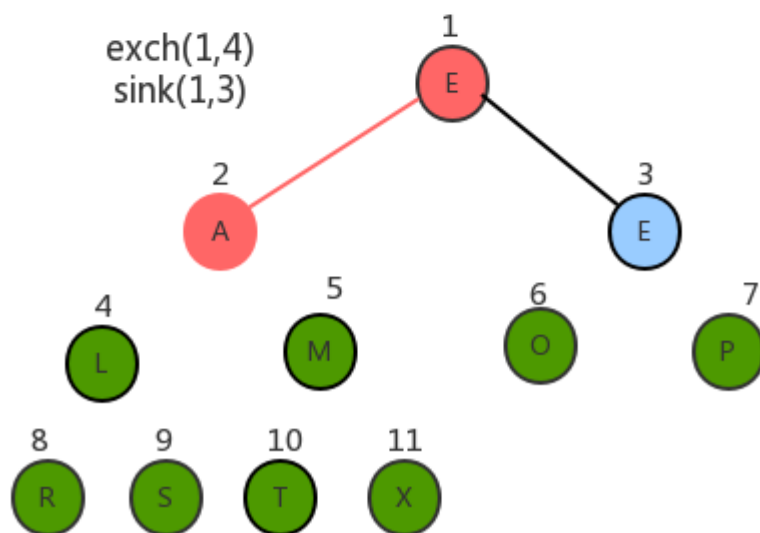
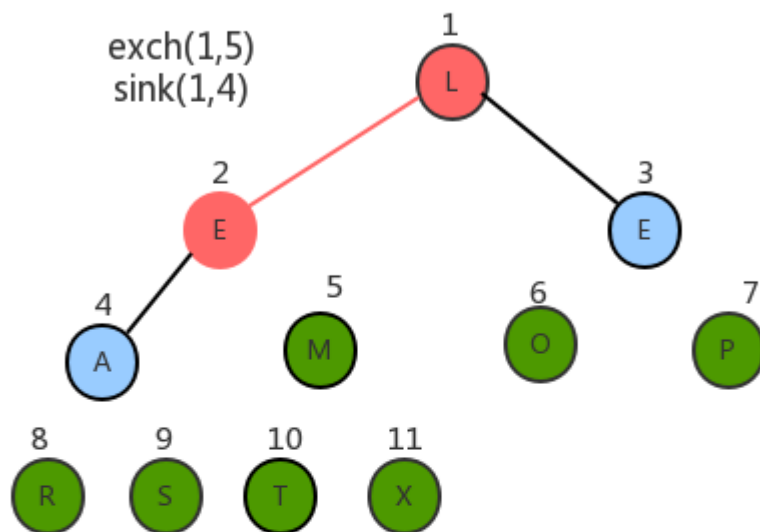
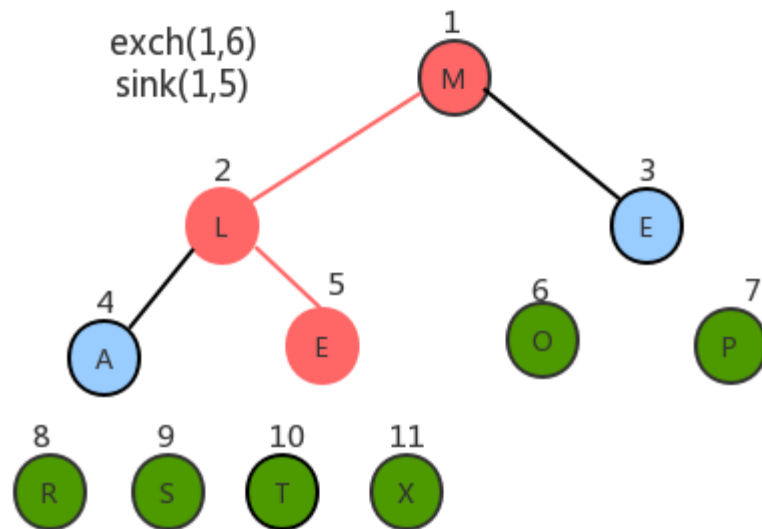
1.4.2 堆排序过程

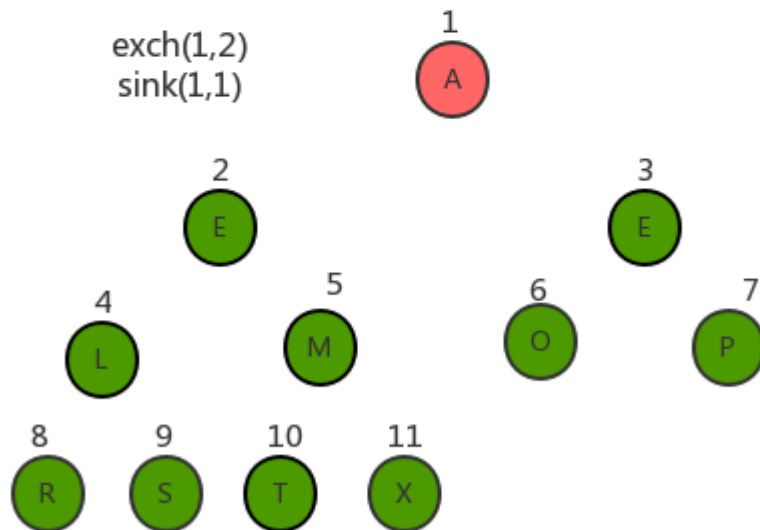
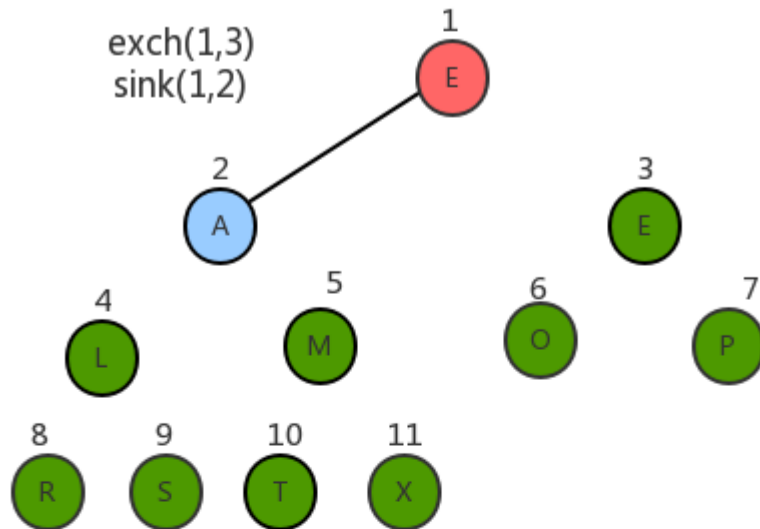
对构造好的堆，我们只需要做类似于堆的删除操作，就可以完成排序。

- 1.将堆顶元素和堆中最后一个元素交换位置；
- 2.通过对堆顶元素下沉调整堆，把最大的元素放到堆顶(此时最后一个元素不参与堆的调整，因为最大的数据已经到了数组的最右边)
- 3.重复1~2步骤，直到堆中剩最后一个元素。









结果，已排序

代码：

```

1 //对排序代码
2 public class HeapSort {
3     //对source数组中的数据从小到大排序
4     public static void sort(Comparable[] source) {
5         //1.创建一个比原数组大1的数组
6         Comparable[] heap = new Comparable[source.length + 1];
7         //2.构造堆
8         createHeap(source, heap);
9         //3.堆排序
10        //3.1定义一个变量，记录heap中未排序的所有元素中最大的索引
  
```

```

11     int N = heap.length-1;
12
13     while(N!=1){
14         //3.2交换heap中索引1处的元素和N处的元素
15         exch(heap,1,N);
16         N--;
17         //3.3对索引1处的元素在0~N范围内做下沉操作
18         sink(heap,1,N);
19     }
20
21     //4.heap中的数据已经有序，拷贝到source中
22     System.arraycopy(heap,1,source,0,source.length);
23
24 }
25
26 //根据原数组source，构造出堆heap
27 private static void createHeap(Comparable[] source, Comparable[] heap) {
28     //1.把source中的数据拷贝到heap中，从heap的1索引处开始填充
29     System.arraycopy(source,0,heap,1,source.length);
30     //2.从heap索引的一半处开始倒叙遍历，对得到的每一个元素做下沉操作
31     for (int i = (heap.length-1)/2; i>0 ; i--) {
32         sink(heap,i,heap.length-1);
33     }
34 }
35
36 //判断heap堆中索引i处的元素是否小于索引j处的元素
37 private static boolean less(Comparable[] heap, int i, int j) {
38     return heap[i].compareTo(heap[j])<0;
39 }
40
41 //交换heap堆中i索引和j索引处的值
42 private static void exch(Comparable[] heap, int i, int j) {
43     Comparable tmp = heap[i];
44     heap[i] = heap[j];
45     heap[j] = tmp;
46 }
47
48 //在heap堆中，对target处的元素做下沉，范围是0~range
49 private static void sink(Comparable[] heap, int target, int range){
50     //没有子结点了
51     while (2*target<=range){
52         //1.找出target结点的两个子结点中的较大值
53         int max=2*target;
54         if (2*target+1<=range){
55             //存在右子结点
56             if (less(heap,2*target,2*target+1)){
57                 max=2*target+1;
58             }
59         }
60
61         //2.如果当前结点的值小于子结点中的较大值，则交换
62         if(less(heap,target,max)){
63
64             exch(heap,target,max);

```

```
64         }
65         //3.更新target的值
66         target=max;
67     }
68
69 }
70 }
71
72 //测试代码
73 public class Test {
74     public static void main(String[] args) throws Exception {
75         String[] arr = {"S", "O", "R", "T", "E", "X", "A", "M", "P", "L", "E"};
76         HeapSort.sort(arr);
77         System.out.println(Arrays.toString(arr));
78     }
79 }
```